microinformatica

Editat sub egida
Comisiei Naţionale de Informatică

INIŢIERE ÎN C++
PROGRAMARE ORIENTATĂ PE OBIECTE

(Ediţie revizuită)

Cluj-Napoca
1993

Autor
Muşlea Ionuţ

Redactor
Poenaru Codruţa

Designer
Derveşteanu Liviu

Editor
prof. univ. Muntean Emil

Tehnoredactare computerizată
Poenaru Codruţa

© Copyright MicroInformatica SRL, 1993

CUPRINS

CUVÎNT ÎNAINTE 7

1. INTRODUCERE ÎN O.O.P. 9
1.1. Scopul O.O.P. 9
1.2. Scurt istoric 10
1.3. Avantaje O.O.P. 12
1.4. Înainte de început 13

2. SCURTĂ INCURSIUNE ÎN C++ 15
2.1. Intrări şi ieşiri. Stream-uri 15
2.2. Funcţii cu parametrii impliciţi. Funcţii cu acelaşi nume . . 16
2.3. Referinţa 19
2.4. NEW şi DELETE 24
2.5. Clase 25
2.6. Constructori şi destructori 30
2.7. "Redefinirea"operatorilor 35
2.8. FRIENDS 37
2.9. Clase derivate 39
2.10. Polimorfism 45

3. CLASE 48
3.1. Introducere 48
3.2. Declararea claselor 48
3.3. Membrii unei clase. "Scope access operator" 52
3.4. Accesarea membrilor unei clase. Pointeri la metode 58
3.5. Alte amănunte privind declararea claselor. Funcţii "inline" . 62
3.6. Membrii statici 68
3.7. Constructori 71
3.7.1. Iniţializarea structurilor de date 71
3.7.2. Proprietăţi specifice constructorilor 74
3.7.3. Tipuri de constructori. Ambiguităţi 76
3.7.4. Conversii 79
3.7.5. Apelul explicit al constructorilor 80
3.7.6. Reducerea numărului de constructori 82
3.7.7. Constructori şi obiecte statice 82
3.7.8. Constructorii claselor încuibărite 83

3.8. Destructori 87
3.8.1. Definire şi caracteristici 87
3.8.2. Destructorii claselor încuibărite 89
3.8.3. Erori deosebite 90
3.9. Friends 91
3.10. Comportarea constructorilor şi destructorilor în situaţii necesitînd alocare dinamică de memorie 95
3.10.1. Crearea şi distrugerea şirurilor de obiecte 95
3.10.2. "Scurtcircuitarea" mecanismului implicit de gestionare a memoriei alocate dinamic 99
3.11. Probleme rezolvate. Probleme propuse spre rezolvare . . . 109
3.11.1. Array-uri de dimensiuni variabile 109
3.11.2. Hash-table pentru numere de telefon 113
3.11.3. Mecanism perfecţionat de gestiune a memoriei . . . 118

4. MOŞTENIREA 127
4.1. Introducere 127
4.2. Clase derivate 128
4.3. Drepturi de accesare a membrilor "moşteniţi" 130
4.4. Constructori şi destructori 138
4.5. Metode virtuale. Utilizarea listelor eterogene 145
4.6. Moştenire multiplă 157
4.7. Problemă rezolvată. Exerciţii propuse spre rezolvare 167

5. ALTE FACILITĂŢI ALE LIMBAJULUI C++ 178
5.1. Funcţii pure şi clase abstracte 178
5.2. Overloading Operators 180
5.3. Operatori unari şi binari 182
5.4. Operatorii "new" şi "delete" 190
5.5. Operatorul "()" 193
5.6. Operatorul "=" 196
5.7. Operatorul "[]" 201
5.8. Operatorul "->" 204
5.9. Conversii 206
5.10. Problemă propusă spre rezolvare 211

6. STREAM-URI 213
6.1. Introducere 213
6.2. Operaţii de intrare şi ieşire 215
6.2.1. Introducere 215
6.2.2. STREAM-uri şi tipuri de date predefinite 216
6.2.2.1. STREAM-uri de intrare 216
6.2.2.2. STREAM-uri de ieşire 218
6.2.3. Alte metode de manipulare a STREAM-urilor 219
6.2.4. Extractori şi insertori definiţi de utilizator 222
6.3. Tratarea erorilor apărute în operaţii cu stream-uri 224
6.4. Operaţii de intrare şi ieşire cu format 226
6.4.1. Indicatorii de stare ai formatului 226
6.4.2. Manipulatori 229
6.5. Fişiere şi stream-uri 232
6.6. Probleme propuse spre rezolvare 236

ANEXA A 237
Lumea minunată a OOP 237

BIBLIOGRAFIE 240

Pentru Codruţa

"Obiectele sînt reprezentări naturale care facilitează înţelegerea şi modificarea unui program."
Joseph Weizenbaum

CUVÎNT ÎNAINTE

Deşi teoria programării orientate pe obiecte era bine fundamentată încă de acum 20 de ani, ideea nu a prins cu adevărat decît în ultimii ani. Ce se ascunde în spatele conceptului de programare obiectuală? Ideea de a oferi programatorului un instrument performant de reprezentare a lumii în calculator. Cum se poate realiza acest lucru? Foarte simplu: în loc să separe iremediabil DATELE de COD, programarea orientată pe obiecte nu face altceva decît să contopească cele două entităţi. Apropiindu-ne acum de C++, putem spune că acesta îmbină calităţile limbajului C cu avantajele programării obiectuale. O foarte bună caracterizare a rezultatului obţinut o face chiar Bjarne Stroustrup, omul ce a pus bazele noului limbaj de programare general, conceput astfel încît să-i facă pe programatorii serioşi să programeze într-o manieră cît mai plăcută.

Prezenta lucrare se adresează în primul rînd cunoscătorilor limbajului C dornici să-şi modernizeze stilul şi concepţia abordării proiectelor informatice. Avînd în vedere faptul că limbajul C++ este un superset al C-ului, pe întreg parcursul lucrării se vor presupune cunoscute noţiunile ce stau la baza programării în C. O parcurgere exhaustivă în cadrul unei singure cărţi a ambelor limbaje ar avea două neajunsuri: pe de o parte expunerea ar fi prea lungă (vezi cele 4 volume TURBO C++ scoase de BORLAND), iar pe de alta, informaţia specifică C++ s-ar dilua şi răspîndi nepermis de mult.

Începătorilor le recomandăm această carte, cu menţiunea că, înainte de parcurgerea ei, este necesară studierea unei cărţi de C clasic.

O notă specială: avînd în vedere că limba oficială a informaticii este engleza şi că - cel puţin pe moment - marea majoritate a termenilor de specialitate nu au o traducere scurtă şi satisfăcătoare în limba română (Ex: MOUSE, SCOPE, BINDING) pe parcursul lucrării vor fi utilizaţi şi termenii englezeşti.

Lucrarea de faţă descrie pe larg cel mai răspîndit dialect al limbajului C++ în lumea PC-urilor: TURBO C++.

Doresc să le mulţumesc pe această cale prietenilor mei Adrian Frits, Călin Caşcaval, Marius Greere şi Horaţiu Guja pentru observaţiile şi sfaturile

preţioase pe care mi le-au dat. Fără ajutorul lor, această carte nu ar fi existat în forma ei actuală.

AUTORUL

1. INTRODUCERE ÎN O.O.P.

1.1. Scopul O.O.P.

Termenul de OOP provine din englezescul "Object Oriented Programming" , care desemnează disciplina programării obiectuale (sau orientată-obiect). Domeniul OOP este deosebit de larg, şi are la bază ideea generoasă de a unifica două concepte de bază: datele aplicaţiei şi codul necesar tratării lor. Pentru a se ajunge la acest deziderat, se crează utilizatorului facilităţi deosebite în direcţia definirii unor tipuri de date proprii şi a unor operatori destinaţi manipulării lor. Acestea se vor comporta aidoma tipurilor de date şi operatorilor standard (întregi, numere reale, caractere, respectiv adunare, scădere ş.a.m.d.). Astfel, după necesităţi, un programator poate crea tipul COMPLEX sau MATRICE, care să opereze cu numere complexe sau matrici. O secvenţă banală în cazul numerelor reale, de genul

```
complex a,b,c;
matrice m,n,o;
...
c=a+b;
m=n*o;
```

poate deveni o realitate, chiar dacă cele şase variabile reprezintă numere complexe şi matrici. Noile tipuri de date ("complex" şi "matrice") se numesc CLASE , variabilele "a" , "b" , "c" , "m" , "n" şi "o" poartă numele de OBIECTE (sau instanţieri ale claselor "complex" şi "matrice"), iar operatorii redefiniţi "+" şi "*" se numesc METODE ale claselor în cauză. Prin unificarea datelor cu codul se înţelege că o CLASĂ conţine atît structurile de date necesare descrierii unui OBIECT, cît şi METODELE (funcţiile) cu ajutorul cărora obiectele în cauză vor fi manipulate.

Este evident că, în acest mod, gradul de abstractizare e mult mai ridicat, iar programele devin mai uşor de înţeles, depanat şi întreţinut.

Datorită modului de abordare fundamental diferit, nu trebuie să vă sperie faptul că, la început, veţi avea unele dificultăţi de adaptare. Cei care au făcut trecerea de la FORTRAN la PASCAL cunosc acea inerentă senzaţie de nesiguranţă la primele contacte cu recursivitatea şi backtracking-ul. În scurt timp, veţi remarca faptul că munca de pînă acum - în sensul de adaptare a unei probleme la modul de lucru al calculatorului - va fi facilitată de marea flexibilitate şi maleabilitate a abordării obiectuale.

1.2. Scurt istoric

În primele limbaje de programare evoluate, cea mai mare parte a atenţiei era acordată descrierii structurilor de date deoarece, practic, prelucrările ce se făceau asupra lor erau destul de simple şi nu necesitau un efort deosebit. Spre exemplu, putem cita cazul COBOL-ului (şi azi foarte răspîndit), în care circa 70% din lungimea codului sursă este ocupată de o descriere a structurilor de date. În timp, odată cu creşterea complexităţii programelor, a apărut necesitatea unei "ordonări" riguroase a muncii. Ca o treaptă superioară, capabilă să rezolve în bună măsură problemele ivite, s-a impus PROGRAMAREA STRUCTURATĂ. De-a lungul anilor, în special datorită creşterii dimensiunii produselor software, şi acest instrument a devenit la rîndu-i depăşit. Sporirea complexităţii programelor aducea după sine dificultăţi reale legate de rigiditatea şi întreţinerea unor programe mamut. Cu alte cuvinte, deşi preţul produselor era în creştere, calitatea lor tindea să scadă.

În căutarea unei soluţii care să scoată informatica din situaţia dificilă în care se afla, s-a pornit de la ideea că "o bună reprezentare a temei de rezolvat este deseori cauza transformării unei problemei complexe într-una deosebit de simplă" (P.H. WILSON, director al laboratorului de inteligenţă artificială de la M.I.T.). În urma diversificării căutărilor, au început să apară limbaje destinate soluţionării problemelor specifice anumitor domenii de activitate. Dintre tendinţele novatore merită să amintim cel puţin două: limbajele avînd la bază noţiunea de "CADRU" ("FRAME") şi cele ce pornesc de la ideea de "ACŢIUNE". În ciuda faptului că ambele sînt destinate inteligenţei artificiale, există totuşi o deosebire ce merită reţinută. Primele implementează OPERAŢII asupra unor MODELE DE ENTITĂŢI, pe cînd celelalte postulează faptul că OBIECTELE nu sînt simple elemente pasive asupra cărora se fac anumite prelucrări, ci, dimpotrivă, menirea OBIECTELOR rezidă în a ACTIVA prelucrările ce le vor suporta ele însele.

În fine, pornind de la noţiunea de CLASĂ, o altă tendinţă a dat naştere aşa-ziselor LIMBAJE ORIENTATE PE OBIECTE. Primele dintre ele cu adevărat demne de acest nume au fost SIMULA (1965) şi SIMULA-2 (1967). În anii '70 a apărut şi celebrul SMALLTALK. Cel mai mare dezavantaj al lor (din punct de vedere al penetrării pieţei) a fost, din cîte se pare, însuşi faptul că au apărut ca limbaje de sine stătătoare, avînd o răspîndire relativ redusă. Din acest motiv, puţini programatori erau dispuşi să abandoneze limbajele consacrate în acele momente, doar de dragul de a lucra obiectual.

Şi, cum diferenţele între punctul de vedere al cercetătorilor şi cel al marilor industriaşi au la bază însăşi deosebirea dintre ARTĂ şi RENTABILITATE, o bună bucată de vreme moda programării obiectuale a fost uitată. În anii 80, în urma acceptării definitive a limbajului C, un colectiv

condus de Bjarne Stroustrup încearcă introducerea conceptului de CLASĂ într-un dialect numit "C with Classes". Ideea prinde contur şi, în 1983, ia naştere prima versiune a noului limbaj: C++. Aproape imediat apar şi partizanii fanatici ai OOP-ului, aflat acum la o a doua tinereţe. Profitînd deci de marea sa popularitate în rîndul softiştilor şi de multitudinea domeniilor de aplicaţie (de la grafică interactivă la proiectarea interfeţelor utilizator şi de la exploatarea reţelelor de calculatoare la tehnicile de proiectare a compilatoarelor) "moftul" devine modă, iar moda certitudine.

De ce acest succes extraordinar? Mai ales din cauza faptului că limbajul C++ nu face nimic altceva decît să-i dea un nou avînt unuia dintre cele mai la modă limbaje ale momentului (este vorba de "C"). Cînd ne gîndim că toţi cei implicaţi în lumea UNIX-ului, şi nu sînt puţini, pot trece direct la OOP profitînd de păstrarea -practic neschimbată- a sintaxei şi regulilor de construcţie, lucrurile încep să se mai lămurească.

Cu toate acestea, problemele nu lipsesc. O declaraţie foarte sugestivă ar fi cea a "inventatorului" limbajului C++: <Utilizatorii au "sărit" pe C++ înainte ca specialiştii să aibă timpul necesar să-i instruiască de aşa manieră încît să-l poată folosi cu randament maxim!>. Ce ascunde această mărturisire? În principal două aspecte:

a. Lipsa timpului fizic a făcut ca formarea cadrelor de specialitate să nu urmeze ciclul optim. Acesta, se presupune că ar consta în a începe învăţarea programării cu un limbaj PUR obiectual, după care să se facă trecerea la programarea structurată. În acest mod, ambele tehnici ar putea fi perfect stăpînite. Vă veţi întreba care este dezavantajul? Răspunsul este destul de simplu. Dacă OOP-ul nu va fi bine înţeles, vor fi inevitabile situaţiile în care, de exemplu, un programator ce nu a învăţat corect PROLOG-ul este tentat tot timpul să simuleze în acest limbaj procedeele de bază învăţate în C. Sau, analog, să utilizeze şi în PASCAL schemele de simulare a recursivităţii deprinse în FORTRAN.
b. Ca o consecinţă directă a punctului "a.", în numărul din octombrie 1991 al revistei "INFO PC" este semnalat faptul că peste 60% din compilatoarele de C++ existente pe piaţă nu sînt folosite decît pentru dezvoltarea de software structurat, şi nu de tip OOP (altfel zis, se lucrează în C pe un compilator C++).

Nu se pot face anticipări asupra modului în care vor evolua tendinţele în "înaltele sfere ale software-ului", dar, nu ar fi de mirare, că, în cîţiva ani, larga popularitate a limbajului "C++" să nu ducă la o reîntoarcere la punctul de plecare: SMALTALK sau, de ce nu, SIMULA.

Şi ce altă confirmare a importanţei programării obiectuale ar fi mai semnificativă, decît faptul că broşurile de angajare (în domeniul software) la IBM sau MicroSoft cer tuturor viitorilor candidaţi cunoaşterea limbajului

1.3. Avantaje OOP

Vom începe prin a defini principalele concepte ce stau la baza OOP, după care ne vom putea da seama mai bine de avantajele lor.

Prin noul concept de ÎNCAPSULARE se ating mai multe obiective: "contopirea" datelor cu codul (în cadrul aşa numitelor CLASE), facilitatea deosebită de localizare a erorilor (întotdeauna cauza se află în "interiorul" unei singure clase) şi de modularizare a problemei de rezolvat (fiecare clasă va rezolva o anumită problemă).

Un alt termen introdus de OOP, MOŞTENIREA, aduce încă un avantaj specific. În urma definirii unei clase, cu un minim de efort şi timp, se pot preciza seturi de clase asemănătoare, avînd totuşi o trăsătură distinctivă. Un exemplu foarte sugestiv ar fi o clasă a mamiferelor, din care fac parte clasa cîinilor şi cea a primatelor. Cîinii se împart la rîndul lor în cockeri, boxeri, dogi, fox-terieri ş.a.m.d. Practic, cîinii şi primatele au atît unele trăsături comune (moştenite ca mamifere), cît şi altele specifice. Pe de altă parte, dogii şi cockerii au trăsături comune atît în calitate de cîini, cît şi ca mamifere. Dacă delimităm corect clasa mamiferelor (nasc pui vii, au sînge cald, îşi alăptează puii), în momentul în care definim cîinii, e suficient să spunem că sînt mamifere pentru ca toate caracteristicile de mai sus să fie subînţelese. Rămîn prin urmare de specificat acele trăsături ce le deosebesc de celelalte mamifere (carnivore, animale domestice, etc.). Întocmai se petrec lucrurile şi în universul claselor C++. În paginile cărţii veţi găsi dovada clară a facilităţilor aduse de către MOŞTENIRE: posibilitatea de extindere a unui program printr-un minim de efort.

La actualul nivel de cunoştinţe vom defini termenul de POLIMORFISM doar intuitiv, urmînd ca la momentul potrivit să dăm toate amănuntele necesare. Ideea de bază constă în faptul că, într-o ierarhie de clase obţinute prin moştenire, o metodă poate avea forme diferite de la un nivel la altul (specifice respectivului nivel din ierarhie). Pentru a lămuri cît mai bine această, problemă vom lua un exemplu concludent tot din lumea animală. Cosiderăm lumea vie drept o clasă generală avînd printre alte caracteristici şi funcţia (în limbaj OOP am spune METODĂ) de HRĂNIRE. Indiferent că e vorba de o lalea, un parameci, o balenă sau un jaguar, toate aceste vietăţi posedă funcţia de HRĂNIRE. Atîta doar că fiecare dintre ele o realizează altfel. Din acest punct de vedere, HRĂNIREA este un proces abstract, dar care în momentul asocierii lui cu o fiinţă vie devine concret. Niciodată cînd spunem că ne hrănim cîinele sau pisica nu ne gîndim să-i expunem la soare şi să-i stropim, ca pe două lalele.

Există şi alte avantaje ce ţin de mecanismul claselor, dar sînt destul de greu de explicat acum. Ele vor fi prezentate pe parcursul cărţii, în cadrul capitolelor corespunzătoare.

Vom încheia această secţiune prin a sublinia avantajele de ordin practic ale programării obiectuale.

În momentul de faţă, piaţa informatică este invadată de adevărate biblioteci şi colecţii de CLASE. Menirea lor este de a permite un coeficient cît mai ridicat de reutilizare a codului scris. Pe de altă parte, pornind de la aceste clase, utilizatorul poate crea prin moştenire alte clase, care să-l ajute să-şi rezolve problema în mod optim. În plus, programatorul are garanţia folosirii unor proceduri scurte, inteligibile şi, nu în ultimul rînd, CORECTE.

Un alt domeniu de utilizare a bibliotecilor de clase este cel al realizării PROTOTIPURILOR unor produse software. Prin PROTOTIP se înţelege un program funcţional, simplificat şi care să dea o imagine clară a produsului final. Utilitatea lor constă în faptul că, într-un timp foarte scurt şi cu un efort minim din partea producătorului, clientul poate avea o imagine destul de clară a produsului final. De ce am afirmat că bibliotecile de clase pot fi utile în realizarea de protitipuri? Pentru că, dacă în cadrul unei aplicaţii oarecare, avem nevoie de un algoritm de recunoaşteri de forme (de exemplu), nu este necesar ca prototipul să conţină un algoritm optim. Este preferabilă utilizarea unei metode deja implementate în cadrul unei biblioteci, şi care să facă acelaşi lucru. În acest mod, prototipul poate fi prezentat într-un interval de timp foarte scurt, după care se vor înlocui doar acele "module" care sînt deficitare din anumite puncte de vedere (consum excesiv de memorie, viteză nesatisfăcătoare, ş.a.m.d.).

Nu trebuie să neglijăm nici facilitarea de către OOP a cooperării între diversele echipe şi/sau persoane lucrînd la unul şi acelaşi proiect.

1.4. Înainte de început ...

... trebuie făcută o precizare: cartea de faţă nu va aborda TURBO C++ din punct de vedere al mediului de programare (IDE, modele de memorie, diverse utilitare, ...), ci doar ca dialect C++. Toate programele prezentate în carte au fost testate cu ajutorul unui compilator TURBO C++ versiunea 2.00, sub sistemul de operare MS-DOS 5.0.

Şi încă trei observaţii:

a. Implicit, orice fişier care conţine cod "C++" va trebui să aibă extensia ".CPP" pentru a fi compilat corect.
b. Toate programele ce utilizează biblioteca grafică BORLAND conţin un

apel de funcţie de genul

```
initgraph(gd,gm,"c:\\tcpp\\bgi");
```

Pentru ca aceste exemple să poată fi executate corect pe calculatorul dumneavoastră, va trebui să aveţi grijă să modificaţi cel de al treilea parametru de aşa manieră încît să exprime calea corectă spre directorul ce conţine fişierele specifice modului de lucru grafic.

2. SCURTĂ INCURSIUNE ÎN C++

Scopul acestui capitol este de a face o trecere în revistă a celor mai importante caracteristici şi concepte introduse de limbajul C++. Nu se doreşte o discuţie exhaustivă, ci doar o vedere de ansamblu asupra a ceea ce se va discuta în amănunt în următoarele capitole. Prin cîteva exemple destul de simple se va iniţia un prim contact al cititorului cu noile concepte. Explicaţiile furnizate în cadrul acestui capitol vor fi de natură să incite la o parcurgere atentă a celorlalte secţiuni, în care gradul de rafinare al informaţiei va creşte substanţial.

Practic, din cuprinsul acestui capitol nu sînt "obligatoriu" de citit decit primele patru secţiuni. Celelalte conţin doar informaţii ce vor fi prezentate mai pe larg în cadrul capitolelor următoare. Ideea de a scrie un capitol "jumătate facultativ" vine în întîmpinarea celor mai cunoscute două tehnici de învăţare: "BOTTOM-UP" (se începe de la zero, şi se va clădi încet, dar sigur, un edificiu ale cărui dimensiuni vor fi percepute doar în finalul activităţii de învăţare) şi "TOP-DOWN" (se începe printr-un prim contact cu problema în toată complexitatea ei, după care se va trece la aprofundarea diverselor componente). Adepţii primului stil vor putea sări liniştiţi peste a doua jumătate a capitolului. Cu toate că oferim posibilitatea alegerii, noi recomandăm cea de a doua abordare, care are avantajul de a permite o corectă încadrare a fiecărui amănunt în contextul general.

2.1. Intrări şi ieşiri. Stream-uri

Aproape toate cărţile de C dau drept prim exemplu de program celebrul "Hello world !", care afişează acest text pe ecran folosind funcţia "printf". O primă dovadă de non-conformism a limbajului C++ este însuşi faptul că reuşeşte cu mult succes înlocuirea clasicelor funcţii "printf" şi "scanf" (cîţi începători nu au avut dificultăţi cu argumentele - pointeri - ai acesteia din urmă!) cu o bibliotecă ce are la bază conceptul de STREAM. Avînd în vedere că acest concept fundamental este descris pe larg în capitolul 6, să vedem doar cum se operează cu el:

```
# include <iostream.h> // header-ul bibliotecii de stream-uri

void main(void){
 int a;
/* declaratia de mai jos "float b;" va fi comentata conform celeilalte
posibilitati C++ de a scrie o linie de comentariu */
 float b;// aceasta este cea de a doua maniera de comentariu C++
```

```
char c;

cout<<"\n\nTastati un intreg : ";
cin>>a;
cout<<"\nTastati un numar real : ";
cin>>b;
cout<<"\nTastati o litera : ";
cin>>c;
cout<<"\n\nAti tastat "<<a<<" , "<<b<<" , "
   <<c<<".\n";
}
```

În urma execuţiei programului pe ecran se va afişa:

Tastati un intreg : 12

Tastati un numar real : 12.34

Tastati o litera : a

Ati tastat 12 , 12.34 , a.

Aşa cum specifică şi comentariul, "iostream.h" este header-ul unei biblioteci de stream-uri de intrare şi ieşire. El vine să înlocuiască pe "stream.h", header-ul bibliotecii de stream-uri corespunzătoare primei versiuni a limbajului C++. O altă observaţie referitoare la aceeaşi primă linie ar fi legată de noua posibilitate de comentariu, şi anume cea utilizînd două slash-uri. Cele două caractere "/" semnalizează începutul comentariului, el continuînd pînă la sfîrşitul liniei respective.

Probabil aţi intuit deja că "cin" şi "cout" desemnează dispozitivele de intrare şi ieşire (implicit tastatura şi ecranul). Operatorii "<<" şi ">>", folosiţi în C pentru shift-ări, sînt utilizaţi de astă dată drept "pune la ..." şi "preia de la ...". După cum vom vedea mai tîrziu, limbajul C++ permite o astfel de redefinire a operatorilor existenţi. Chiar dacă un operator este redefinit, se vor păstra şi vechile lui sensuri. Atît că, în funcţie de operanzii care-l apelează, se va efectua o operaţie sau alta.

2.2. Funcţii cu parametrii impliciţi. Funcţii cu acelaşi nume

Începem această secţiune prin a relua o afirmaţie făcută în cuvîntul introductiv. DEOARECE LIMBAJUL C++ ESTE UN SUPER-SET AL

LIMBAJULUI C, TOATE DECLARAŢIILE ŞI DEFINIŢIILE DE VARIABILE ŞI FUNCŢII, CARACTERISTICE DIALECTULUI TURBO C, VOR RĂMÎNE ÎN CONTINUARE VALABILE. Cum ne-am propus doar prezentarea caracteristicilor C++, nu vom explica în cele ce urmează nici un fel de instrucţiuni care, asemeni definirii funcţiei "exemplu", nu necesită decît cunoaşterea limbajului C:

```
int exemplu(int p)
{
 int i;
     for(i=0;i<p;i++)
     printf("\n\t i=%d",i);
     return p/2;
}
```

Vom continua prin a analiza cu atenţie exemplul de mai jos:

```
# include <iostream.h>

void functie(int a=123,double b=456.789,
              char * c="program")

{
 cout<"\na="<<a<<" b="<<b<<" c="<<c;
}

void main(void)
{
 functie(321,987.654,"subrutina");
 functie(321,987.654);
 functie(321);
 functie();
}
```

În urma execuţiei secvenţei de instrucţiuni, pe ecran vor apărea următoarele şiruri de caractere:

a=321 b=987.654 c=subrutina
a=321 b=987.654 c=program
a= 321 b= 456.789 c= program
a=123 b=456.789 c=program

Probabil că intuiţi deja despre ce este vorba: prin declaraţia

```
void functie(int a=123,double b=456.789,
              char * c="program");
```

compilatorul este atenţionat că orice apel al funcţiei "functie" poate avea între

zero şi trei parametri actuali. În caz că numărul lor este mai mic de trei, cei lipsă vor lua IMPLICIT valorile specificate în lista de parametri formali. Dar atenţie! Această facilitate are şi ea limitele ei: indiferent de numărul parametrilor actuali ai funcţiei, ordinea şi tipul lor sînt întotdeauna aceleaşi! Ba chiar mai mult, începînd cu primul parametru transmis implicit, TOŢI cei care-i urmează în listă nu mai pot fi transmişi DECÎT cu valori implicite!

"Functions overloading" - în traducere liberă "suprapunerea" ori "supraîncărcarea" funcţiilor - este un alt concept fundamental în C++. Graţie lui, utilizatorul poate să declare funcţii cu acelaşi nume, dar avînd prototipuri diferite (altfel spus, este vorba de funcţii cu acelaşi nume, dar cu semnături diferite). Aceste funcţii pot fi distinse prin faptul că vor diferi fie prin tipurile parametrilor formali, fie prin tipul valorii returnate. Vom începe prin a da un exemplu, pe care-l vom comenta în continuare.

```
# include <iostream.h>
# include <string.h>

// in TURBO C++ declaratia ce urmeaza nu este obligatorie
 overload a;

// prototipurile celor doua functii avind acelasi nume

void a(void);
int a(char *);

// definitiile celor doua functii

void a(void){
     cout<<"\n Nu exista parametri !\n";
 }

 int a(char *s){
     cout<<"\n Parametrul este pointer la
           stringul : "<<s;
     return strlen(s);
 }

void main(void){
 a();
 cout<<"\n Lungimea stringului = "<<a("asdfe");
}
```

Pe ecran se vor afişa următoarele texte:


```
Nu exista parametri !

Parametrul este pointer la stringul : asdfe
Lungimea stringului = 5
```

La versiunea iniţială (AT&T 1.0) a lui Bjarne Strostroup, în C++ era necesară utilizarea cuvîntului cheie "overload", urmat de numele funcţiei respective. În noile implementări, acest lucru nu mai este impus. Totuşi, recomandăm folosirea unei astfel de declaraţii, în ideea că un program trebuie scris cît mai explicit. Pentru că, nu uitaţi, programele trebuie să fie uşor de înţeles atît pentru cel care le-a făcut, cît şi pentru o altă persoană ce ar urma să-i continue munca!

În situaţia de faţă, utilitatea unui astfel de exemplu poate fi îndoielnică. Dacă însă dorim lărgirea bibliotecii matematice a limbajului C++ cu noi funcţii (de exemplu ridicarea la putere), poate prezenta un cert interes existenţa unor funcţii cu acelaşi nume ("PUTERE"), dar care operează cu diverşi parametrii (numere complexe sau raţionale).

Exemplu:

```
typedef struct rational1{
 float numarator,numitor;
 } rational1;

typedef struct complex1 {
 float p_reala , p_imaginara;
 } complex1;

overload PUTERE;
rational1 * PUTERE(rational1 * , int );
complex1 * PUTERE(complex1  *,int );
```

2.3. Referinţa

În absolut toate domeniile informaticii se pune foarte acut problema eficienţei. Ea comportă două aspecte: unul legat de spaţiul de memorie utilizat şi un altul legat de timpul de execuţie al programelor. În legătură cu acesta din urmă, este unanim cunoscută preferinţa pentru limbajul de asamblare a tuturor programatorilor interesaţi de economisirea fiecărei fracţiuni de secundă. Dar, de la acest aspect pînă la dimensiunile pe care le ia astăzi fenomenul, este o cale foarte lungă. Vom da două exemple elocvente în acest sens.

Specialiştii de la MicroSoft au tras concluzia că folosirea specificatorului

"pascal" în faţa funcţiilor scrise în C scurtează codul generat cu 7%. Acest cîştig se obţine prin faptul că, în convenţia PASCAL, stiva este gestionată de către rutina apelată, nu de către cea apelantă.

Al doilea exemplu este rezultatul studiilor lui Bjarne Strostroup. Acesta afirmă că timpul consumat de un apel de funcţie de tipul "alloc" este atît de mare încît este mai rentabil ca, în cazul unor alocări repetate de memorie pentru date de dimensiuni reduse, să se proiecteze un "gestionar" al memoriei. Acesta va aloca la un moment dat o cantitate mai mare de memorie, din care să "livreze", la comandă, cantităţile necesare.

Revenind, să amintim că, în cazul unor apeluri de funcţii care au drept parametri obiecte de mari dimensiuni, este foarte necconomică punerea acestora pe stivă la apelul funcţiei. O economie semnificativă se poate obţine prin utilizarea pointerilor. Aceasta are, însă, două dezavantaje:

a. este destul de puţin elegantă deoarece, în unele cazuri, este mult mai sugestivă operarea asupra obiectelor decît cea asupra pointerilor la ele;
b. există cazuri speciale în C++ cînd o astfel de soluţie este contraindicată.

De aceea s-a făcut apel la un nou tip de dată, numit tipul REFERINŢĂ. Acesta implementează perfect conceptul de "transmitere a parametrilor prin referinţă". O referinţă reprezintă, de fapt, numele unui obiect. Dacă avem tipul de dată "T", prin "T&" se va înţelege o referinţă la un obiect de tipul "T". Orice referinţă trebuie iniţializată, după care devine pur şi simplu un alt nume (un "alias") al obiectului cu care a fost iniţializată.

Pentru a exemplifica cît mai bine eficienţa utilizării referinţelor, vom analiza două exemple: unul care foloseşte şi unul care nu foloseşte tipul referinţă. Programul de mai jos nu va folosi tipul referinţă, pe cînd cel care-i urmează îl va utiliza:

```
# include <iostream.h>
# define MAX_NUME 25

 typedef struct _persoana {
    char nume[MAX_NUME],prenume_1[MAX_NUME],
           prenume_2[MAX_NUME],initiala_tatalui;
    int virsta,cod;
    float inaltime;
 } persoana;

void tipareste_persoana(persoana p){
 cout<<p.nume<<" "<<p.prenume_1<<" "<<p.prenume_2;
 cout<<"    virsta : "<<p.virsta<<" , cod ="<<
     p.cod<<" , inaltimea = "<<p.inaltime<<"\n\n";
}

void main(void){
 persoana James_Bond={"Bond","James","",'K',40,
                          007,1.88};
 tipareste_persoana(James_Bond);
 persoana Ion_Ion=James_Bond;
 tipareste_persoana(Ion_Ion);
}
```

Pe ecran se va tipări:

Bond James virsta : 40 , cod = 7 , inaltimea = 1.88

Bond James virsta : 40 , cod = 7 , inaltimea = 1.88

Pentru înţelegerea programului, trebuie subliniat că instrucţiunile de genul

```
persoana Ion_Ion=James_Bond;
```

realizează iniţializarea printr-o copie bit cu bit a celei de a doua structuri în cadrul primeia.

Structura "persoana" ocupă un număr semnificativ de octeţi. Dacă se doreşte tipărirea unui şir de 1000 de persoane, timpul ocupat de transmiterea succesivă a celor 1000 de structuri ca parametri creşte nepermis de mult. Pentru a scăpa de acest inconvenient, vom folosi un al doilea program, care utilizează drept parametri referinţe la structuri de tip "persoana".

Iată acum cel de-al doilea program, în care se va utiliza tipul referinţă:

```
# include <iostream.h>
# define MAX_NUME 25

 typedef struct _persoana {
    char nume[MAX_NUME],prenume_1[MAX_NUME];
    char prenume_2[MAX_NUME],initiala_tatalui;
    int   virsta,cod;
    float inaltime;
 } persoana;
 persoana sir_persoane[20];

void tipareste_persoana(persoana& p){
 cout<<p.nume<<" "<<p.initiala_tatalui<<". "<<
    p.prenume_1<<" " <<p.prenume_2<<"\n";
 cout<<" virsta : "<<p.virsta<<" , cod = "<<
    p.cod<<" , inaltimea = "<<p.inaltime<<"\n\n\n";
```

```
}

void actualizeaza_virsta(persoana& refPersoana)
{
 refPersoana.virsta++;
}

persoana& cauta_persoana_de_50_ani(void)
{
 for(int i=0;i<20;i++)
    if(sir_persoane[i].virsta==50)
         return sir_persoane[i];
}

void main(void)
{
 cout<<"\n\n\n\n\nProgram demonstrativ :
   REFERINTE\n\n\n";
 persoana James_Bond={"Bond","James","",'K',40,
                      007,1.88};

 persoana& ref_pers1=James_Bond;
 tipareste_persoana(ref_pers1);

 persoana Ion_Ion=James_Bond;
 persoana& ref_pers2=Ion_Ion;
 tipareste_persoana(ref_pers2);

 actualizeaza_virsta(ref_pers2);
 tipareste_persoana(ref_pers2);

 tipareste_persoana(James_Bond);

 persoana aa={"Iftode","Ilie","",'K',40,007,1.88};

 for(int i=0;i<20;i++)
 {
    // initializare referinta
    persoana& ref_pers3=sir_persoane[i];
    sir_persoane[i]=aa; // initializare element din sir
    for(int j=-1;j<i;j++)
       actualizeaza_virsta(ref_pers3);
 }
```



```
 cout<<"\n\nPersoana de 50 de ani : \n";
 tipareste_persoana(cauta_persoana_de_50_ani());
}
```

În urma execuţiei programului pe ecran se va afişa:

Program demonstrativ: REFERINTE
Bond K. James
virsta : 40 , cod = 7 , inaltimea = 1.88

Bond K. James
virsta : 40 , cod = 7 , inaltimea = 1.88

Bond K. James
virsta : 41 , cod = 7 , inaltimea = 1.88

Bond K. James
virsta : 40 , cod = 7 , inaltimea = 1.88

Persoana de 50 de ani :
Iftode K. Ilie
virsta : 50 , cod = 7 , inaltimea = 1.88

De astă dată, în funcţia "tipareste_persoana" nu va fi transmisă întreaga structură, ci doar adresa ei. Parametrul "refPersoana" va referi structura de tipul "persoana" care apărea pe post de parametru formal. De remarcat că membrii structurii se referă prin ".", nu prin "->" (să nu uităm că o referinţă la un obiect nu este altceva decît un ALT NUME desemnînd exact aceeaşi instanţiere cu care a fost iniţializată).

După cum se va vedea în urma rulării programului, acţiunile ce se vor executa în "actualizeaza_virsta" vor fi păstrate în cadrul obiectelor şi în urma părăsirii acestei funcţii. Acesta este un al doilea aspect al utilizării referinţelor: se lucrează direct asupra obiectului referit. De remarcat că şi în cazul utilizării referinţelor, iniţializările de obiecte prin atribuire au loc exact în maniera descrisă mai sus (adică tot prin copiere bit cu bit).

În cadrul ultimelor două proceduri se remarcă un lucru foarte interesant. Restricţia din limbajul C, legată de obligativitatea declarării tuturor variabilelor locale la începutul procedurii, nu mai e valabilă (vezi locul în care sînt declarate "aa" şi "ref_pers3"). Mai mult chiar, se vede că, în două rînduri, contorul întreg "i" este definit în cadrul instrucţiunii "for". Totuşi se mai cuvine făcută o observaţie: instrucţiunile de genul

```
for(int i=0;.....)
```

sînt echivalente cu

```
int i;
for(i=0;...)
```

Altfel spus, o secvenţă

```
for(int i=0;...)
{
   // ....
}
for(int i=0;...)
```

va provoca eroarea de "redefinirea variabilei i". Am atras atenţia asupra acestui fapt deoarece multă lume este tentată să considere că declararea lui "i" în cadrul unui "for" îl face să devină o variabilă locală acestuia. Deci nu uitaţi: domeniul de existenţă al lui "i" ia sfîrşit la închiderea blocului ce-l conţine pe "for".

2.4. NEW şi DELETE

Programele scrise în TURBO C neobiectual foloseau pentru alocarea dinamică de memorie familiile de funcţii "free" şi "malloc", care erau destul de greoaie în folosire. Ele au fost păstrate în TURBO C++ doar pentru menţinerea compatibilităţii cu versiunile precedente. În locul lor, se pot folosi cei doi operatori creaţi în acest scop. Primul, "new", serveşte la alocarea dinamică a memoriei, iar cel de al doilea - "delete" - la eliberarea ei.

Operatorul "new" va returna un pointer la zona de memorie alocată dinamic. În cazul în care nu există suficientă memorie disponibilă, alocarea nu va avea loc. Acest fapt este semnalizat prin returnarea unui pointer NULL. De aceea, se recomandă că, după fiecare utilizare a lui "new", să se testeze valoarea returnată. În caz de eşec al operaţiei de alocare, este bine să se tipărească un mesaj de avertizare şi să se încheie execuţia programului. Altfel există riscul de a se obţine efecte dezastruoase.

Operatorul "delete" va elibera zona de memorie la care pointează argumentul său.

În programul de mai jos se poate vedea un exemplu de testare a corectitudinii alocării dinamice de memorie.

Exemplu:

```
# include <iostream.h>
# include <process.h>
```



```
# define nl "\n"

 typedef struct _pereche {
    int a,b;
    struct _pereche *next;
 } pereche;

// Testeaza valoarea parametrului actual. Daca el este NULL
// inseamna ca ultima operatie de alocare dinamica de memorie
// s-a soldat cu un esec. Din acest motiv se afiseaza un mesaj
// de avertizare si se provoaca oprirea executiei programului.
void testare_pointer(void * pointer){
 if(pointer==NULL)
 {
    cout<<"\n\nMemorie insuficienta !!!\n";
    exit(1);
 }
}

void main(){
 pereche *p1,*p2;

 p1=new pereche;   // se aloca memorie pentru o structura
 testare_pointer((void *) p1); // test alocare corecta
 p2=new pereche[10]; // se aloca memorie pentru un
                     // sir cu 10 elemente
 testare_pointer((void *) p2); // test alocare corecta
 p2[2].a=10;
 cout<<nl<<p2[2].a<<nl;
 delete p1; // se elibereaza memoria alocata pentru o structura
            // ...
}
```

O mare parte din exemplele cărţii vor utiliza operatorul "new". În cele mai multe din cazuri, nu s-a testat valoarea returnată de acesta. Omisiunea este INTENŢIONATĂ şi are două scopuri: pe de o parte se evită încărcarea excesivă a exemplelor, iar pe de alta nu se distrage atenţia de la scopul secvenţei de instrucţiuni.

2.5. Clase

În cadrul acestei secţiuni vom descrie succint una dintre cele mai impor-

tante noutăţi aduse de limbajul C++, şi anume conceptul de clasă. Cu ajutorul claselor se realizează un vechi deziderat al programatorilor: unificarea datelor şi a procedurilor.

Pentru a înţelege cît mai bine ideea ce stă la baza conceptului de clasă, vom analiza pentru început ideea de tip de dată, sub forma în care există ea în limbajul C. Spre exemplu, dacă ne referim la numerele întregi ("int"), vom remarca faptul că acestea sînt descrise de două trăsături: în primul rînd spaţiul de memorie în care numerele sînt reprezentate (şi care variază de la un calculator la altul), iar în al doilea rînd un set de operatori cu ajutorul cărora întregii pot forma expresii aritmetice sau logice. Programatorul nu este interesat de soluţia folosită de compilator pentru rezolvare a celor două aspecte, ci doar de modul în care poate utiliza respectivul tip de dată.

În ciuda facilităţilor de limbaj cu totul remarcabile, C-ul nu poate (şi nici nu are cum) să fie dotat cu toate tipurile de date şi operaţii pe care şi le-ar putea dori diverşii utilizatori. O rezolvare elegantă a acestui inconvenient este adusă de C++ prin conceptul de clasă. Cu ajutorul claselor, un programator îşi poate defini orice tip de dată dorit, împreună cu setul necesar de operaţii. Toate aceste informaţii vor fi reunite (încapsulate) într-o CLASĂ. Astfel, prin clasă vom înţelege un nou tip de date, definit de utilizator. Clasele pot fi folosite întocmai ca şi tipurile de date predefinite. Făcînd abstracţie de modul de realizare a celor spuse pînă acum, ne putem închipui foarte bine următoarea secvenţă de instrucţiuni C++:

```
complex a(10,1),b(2,3),c;
rational d(4,3);
c=a*b;
d=d*2;
```

În acest mod utilizatorul va opera cu numere complexe şi raţionale exact ca şi cum ar folosi tipurile predefinite "int"sau "float".

Înainte de a da un exemplu concret de implementare a unei clase, vom explica sumar modul de creare a unui PROJECT în mediul TURBO C++. Facem acest lucru deoarece la noul produs al firmei BORLAND o astfel de operaţie nu mai are loc manual, ci se efectuează cu sprijinul unor funcţii speciale ale meniului. Celor care nu cunosc tehnica de lucru cu PROJECT-uri le vom recomanda să studieze un manual de utilizare TURBO C. În cîteva cuvinte totuşi, despre ce este vorba. În cazul programelor de dimensiuni mai mari, scrierea codului în cadrul unui singur fişier e neplacută şi neeconomicoasă: de fiecare dată se vor compila toate liniile programului, chiar dacă s-a modificat un simplu comentariu. De aceea, se recomandă împărţirea programului în module mai scurte, care să fie scrise în fişiere separate. Aceste fişiere vor fi compilate independent, iar prin link-editarea lor în comun (în cadrul unui PROJECT) va lua naştere un singur program executabil. Avantajele sînt multiple: pe de o parte un astfel de produs software este mai uşor de gîndit şi întreţinut, iar pe de alta, în cadrul acţiunii de modificare a unuia dintre module, nu va fi recompilat decît modulul în cauză.

Revenind la mecanismul de creare/actualizare a PROJECT-urilor TURBO C++, acesta este implementat în cadrul unor funcţii din meniul de bază. Astfel, în urma selectării din meniul principal a opţiunii PROJECT, se va putea fie crea, fie încărca un nou PROJECT (opţiunea OPEN PROJECT). În cadrul acestuia se vor putea folosi opţiunile INSERT şi DELETE, cu ajutorul cărora se adaugă/şterg fişiere la/din PROJECT-ul selectat. O dată creat un PROJECT, prin comanda RUN se vor compila şi link-edita toate fişierele sale, creîndu-se un singur program executabil.

Am făcut paranteza de mai sus deoarece în cadrul acestei secţiuni vom prezenta un prim exemplu scris modular, care va servi drept obiect de studiu atît pentru noţiunea de CLASĂ, cît şi pentru cea de PROJECT.

Începem cu un program compus din trei fişiere: STTNG_CL.DAT, SETTINGS.CPP şi MAIN_SET.CPP. Pentru a obţine din cele trei fişiere un program executabil este necesară crearea unui PROJECT (să-i zicem SETTINGS.PRJ), care să asigure link-editarea corectă a lui SETTINGS.CPP şi MAIN_SET.CPP (STTNG_CL.DAT va fi inclus ca header în celelalte două fişiere). Să vedem deci ce conţine fişierul STTNG_CL.DAT.

```
// DECLARAREA CLASEI "settings"
# include <graphics.h>

class settings {
   private:
   int cul,bkcul; // culoare de scriere si de fond
   struct textsettingstype txt; // atributele de scriere
                                // a textelor
   struct fillsettingstype fll; // atributele de umple-
                                // re a figurilor
   public:
   void put_settings(void);
   void get_settings(void);
};
```

Odată făcut acest lucru, să încercăm să accentuăm două definiţii. O CLASĂ este un tip abstract de date, definit de către programator. Un OBIECT este o variabilă declarată ca fiind de tipul CLASĂ. Cu alte cuvinte, un OBIECT este o INSTANŢIERE a unei CLASE.

Clasa pe care tocmai am declarat-o poartă numele "settings" (acesta este numele noului tip de dată pe care l-am definit). Ea are două secţiuni: cea

privată şi cea publică. În cazul nostru (foarte particular, de altfel) prima conţine doar declaraţii de variabile, iar a doua numai prototipuri de funcţii. Acest fapt se datorează unei pure întîmplări, în ambele secţiuni putînd figura, fără restricţii, atît variabile cît şi funcţii. Secţiunea privată păstrează (invizibil pentru utilizatorul clasei) informaţii folosite de către implementatorul clasei, pe cînd cea publică reprezintă interfaţa dintre clasă şi potenţialul ei utilizator. Pentru o mai bună înţelegere, vom face o analogie cu tipul de date "int" din C. În acest caz atît octeţii în care e memorat întregul, cît şi operatorii existenţi (+, -, * ş.a.m.d.) se află la dispoziţia utilizatorului, deci ele sînt publice.

O precizare care va lămuri multe neclarităţi: un "struct" din limbajul C clasic poate fi privit ca o clasă cu toate variabilele publice şi neavînd nici o funcţie membru. Am definit astfel şi funcţiile membru: acele funcţii ce sînt declarate în cadrul unei clase. Şi fiindcă tot a venit vorba de structuri, să spunem că, în C++, şi lor li se pot adăuga funcţii membru. În acest moment este clar că o secvenţă ca cea de mai jos nu mai reprezintă o enigmă pentru dumneavoastră.

```
class _clasa {
 // ...
 public : int i;
 void do_it(void) { cout<<i; };
 // ...
};

// ...

_clasa my_class; // declararea obiectului "my_class"
my_class.i=5;       // setarea membrului "i" la 5
my_class.do_it(); // apelarea functiei membru "do_it"
```

Să reluăm în cuvinte cele cîteva idei puse în evidenţă de acest scurt exemplu. Atît variabilele cît şi funcţiile membru ce apar în secţiunea publică a unei clase pot fi utilizate în orice punct al programului. Există totuşi o restricţie: ele pot fi accesate doar prefixate de către un nume de obiect (instanţiere a clasei respective). Operatorul ce face posibilă selectarea membrilor unui obiect este punctul ("."), întocmai ca în cazul structurilor din C. În schimb, toate variabilele şi metodele private nu vor putea fi accesate DECÎT de către alte metode ale clasei în cauză (trebuie să subliniem că în terminologia OOP "metoda" este sinonimă cu "funcţie membru").

În continuare, vom analiza fişierul "SETTINGS.CPP", care conţine descrierea funcţiilor membru ale clasei "settings".


```
/* Salveaza/restaureaza atributele modului de lucru grafic */
# include "sttng_cl.dat" // e inclus ca header

// Restaureaza atributele modului de lucru grafic salvate cu
// get_settings()
void settings::put_settings(void)
{
 settextstyle(txt.font,txt.direction,txt.charsize);
 settextjustify(txt.horiz,txt.vert);
 setfillstyle(fll.pattern,fll.color);
 setcolor(cul);
 setbkcolor(bkcul);
}

// Salveaza atributele curente de lucru in mod grafic
void settings::get_settings(void)
{
 gettextsettings(&txt);
 getfillsettings(&fll);
 cul=getcolor();
 bkcul=getbkcolor();
}
```

Din cîte se vede, fişierul STTNG_CL.DAT este inclus sub formă de header. El a fost conceput ca fişier separat tocmai pentru a feri utilizatorul de munca plicticoasa de a tasta informatiile incluse in acesta ori de cite ori intr-unul dintre modulele unui PROJECT este utilizat un astfel de obiect. Pentru toţi cei care cunosc nucleul grafic al firmei BORLAND, scopul în care a fost creată clasa SETTINGS este evident. Ea realizează un mecanism simplu şi comod de salvare/restaurare a unor caracteristici specifice modului de lucru grafic (culoare de scriere/fond, font, direcţie de scriere, dimensiune caracter, şablon de umplere a figurilor etc.).

Vom mai sublinia că, la definirea funcţiilor membru, numele acestora este prefixat de numele clasei urmat de operatorul "::". Această construcţie indică faptul că este vorba de funcţia membru a unei clase anume. O astfel de măsură de prevedere este necesară deoarece în cadrul unui program putem avea mai multe clase care posedă funcţii membru cu acelaşi nume. De asemenea, poate exista o funcţie membru avînd acelaşi nume ca şi o funcţie nemembru.

În fine, vom prezenta fişierul MAIN_SET.CPP, care prezintă demonstrativ calităţile clasei SETTINGS.

```
# include <conio.h>
# include "sttng_cl.dat"//header ce descrie clasa SETTINGS;
                        //prezenta lui este necesara in caz
```

```
                        //ca se doreste folosirea de obiecte
                        //de acest tip

void main()
{
 int gd,gm;

 detectgraph(&gd,&gm);
 initgraph(&gd,&gm,"\\tcpp\\bgi");
 outtextxy(50,100,"HELLO !");

 getch();
 cleardevice();

 settings my_settings;
 my_settings.get_settings();

 settextstyle(GOTHIC_FONT,HORIZ_DIR,4);
 outtextxy(100,100,"HELLO !");
 settextstyle(SMALL_FONT,VERT_DIR,7);
 outtextxy(500,100,"HELLO !");

 my_settings.put_settings();
 getch();

 cleardevice();
 outtextxy(100,100,"HELLO !");

 getch();
 closegraph();
}
```

2.6. Constructori şi destructori

Să revenim la analogia între o clasă oarecare şi tipul de date "int". În cele expuse atunci, am precizat o serie de asemănări. De astă dată ne vom pune o problemă de altă natură: cine şi cînd alocă cei doi octeţi (în general acesta este numărul de bytes pe care se reprezintă un întreg) necesari memorării fiecărei variabile întregi? Răspunsul nu ne interesează din punct de vedere al corectitudinii, ci prin prisma sugestivităţii sale. De aceea ne putem închipui foarte bine că în momentul întîlnirii unei declaraţii de genul:

```
int i;
```


există o funcţie specială "int()" care rezervă numărul de octeţi necesari. În cazul în care se întîlneşte o formă de genul:

```
int j=9;
```

se va apela "int(9)", care, pe lîngă alocarea octeţilor necesari, îi mai şi iniţializează cu valoarea 9.

Un mecanism absolut similar trebuie să existe şi pentru eliberarea memoriei respective în momentul în care se părăseşte domeniul de existenţă al variabilelor. Dacă celor două funcţii le vom da numele generice de "constructor" şi respectiv "destructor", putem spune că am definit alte două concepte esenţiale ale programării obiectuale.

De remarcat că o clasă poate avea mai mulţi constructori. De exemplu, chiar în analogia noastră cu numerele întregi, avem de-a face cu doi constructori. Unul nu are nici un parametru (rezervă doar spaţiu), pe cînd al doilea are un parametru ce constituie valoarea iniţială a variabilei. De notat că, dacă o clasă este prevazută cu constructori, la fiecare declarare a unui obiect de acest tip va avea loc apelul unuia dintre constructori. Tot acum trebuie subliniat că ÎNTOTDEAUNA constructorul va purta numele clasei în cauză. La fel şi destructorii, atît ca numele respectiv va fi precedat de un caracter " ~ ".

În continuare, vom da un exemplu de clasă avînd atît constructor, cît şi destructor.

```
// - clasa STACK implementeaza o stiva ale carei elemente sint
//   siruri de caractere

// - stiva este simulata printr-o lista simplu inlantuita

# include <iostream.h>
# include <process.h>
# include <string.h>
# include <stdio.h>

class stack {
 struct _stiva {
    char info[20];
    struct _stiva *next;
 } *head;      // pointer la inceputul stivei de stringuri

public:
 stack(void);    // CONSTRUCTOR !!!
 ~stack(void);   // DESTRUCTOR !!!
```

```
 void pop(void);
 void push(char *);
 void print_stack();
};

// Initializeaza lista ca fiind vida
inline stack::stack(void)
{
 head=NULL;
}

// Distruge stiva (lista simplu inlantuita)
stack::~stack(void)
{
 struct _stiva *a; // var.tampon
 if(head!=NULL)
 do
 {
     a=head->next;
     delete head;
     head=a;
 }
 while (a!=NULL);
}

// Tiparestestiva
void stack::print_stack(void)
{
 struct _stiva *a;     // var. tampon
 a= head;
 if(head!=NULL) {
     cout<<"\n STIVA CONTINE :\n\n";
     do {
         cout<<a->info<<"\n";
         a=a->next;
     }
     while (a!=NULL);
     cout<<"\n";
 }
 else cout<<"\n STIVA ESTE VIDA !";
}
```



```
// Descarca un element de pe stiva (elimina primul element din lista)
void stack::pop(void)
{
 struct _stiva *a;
 if (head!=NULL) {
     a=head;
     head=head->next;
     delete a;
 }
 else cout<<"Nu se poate executa \"pop\" dintr-o
          stiva goala !";
}

// Pune un element pe stiva (adauga la inceputul listei un nou
// element)
void stack::push(char *info)
{
 struct _stiva *a=new struct _stiva;
 if (a==NULL)
 {
     cout<<"Memorie insuficienta in STACK::PUSH\n";
     exit(1);
 }
 strcpy(a->info,info);
 a->next=head;
 head=a;
}

void main(void)
{
 stack my_stack;

 my_stack.push("Aceasta");
 my_stack.push("este");
 my_stack.push("stiva");

 my_stack.print_stack();
 my_stack.pop();
 my_stack.print_stack();

 my_stack.pop();
 my_stack.pop();
 my_stack.pop();
```

```
 my_stack.print_stack();
 my_stack.push("Sirsit");
 my_stack.print_stack();
}
```

În urma execuţiei programului, pe ecran se vor tipări următoarele date:

```
STIVA CONTINE:

stiva
este
Aceasta

STIVA CONTINE:

este
Aceasta

Nu se poate executa "pop" dintr-o stiva goala !
STIVA ESTE VIDA !
STIVA CONTINE:

Sfirsit
```

La implementarea constructorului s-a putut vedea următoarea construcţie:

```
inline stack::stack(void)
{
    head=NULL;
}
```

Cuvîntul cheie "inline" este folosit pentru a indica faptul că funcţia definită astfel va fi expandată în toate locurile unde este apelată. Altfel spus, e vorba de o modalitate modernă şi elegantă de a defini macrouri-funcţie. Prefixarea cu "inline" are sens doar pentru funcţiile foarte scurte, la care este mai costisitor (în timpi de execuţie) mecanismul de apel al procedurii decît executarea corpului propriu-zis al acesteia. Un astfel de caz este şi cel de faţă: corpul funcţiei nu conţine decît o simplă iniţializare. De remarcat şi lipsa oricărei specificări de tip returnat de către constructor.

Procedura "stack::~stack" este destructorul stivei. În momentul în care domeniul de existenţă al variabilei este depăşit, se va apela această funcţie în scopul eliberării memoriei alocate dinamic.

Ar mai fi de analizat următoarea structură:


```
void stack::print_stack(void)
{
 struct _stiva *a; //var.tampon
 a= head;
 // ...
}
```

în care funcţia membru "print_stack" operează cu variabila privată "head". În caz că avem mai multe stive S1, S2 şi S3, pentru fiecare din apelurile

```
S1.print_stack();
S2.print_stack();
S3.print_stack();
```

a "head" trebuie să aibă valoarea specifică stivei în cauză. Vă veţi întreba, pe bună dreptate, cum se realizează acest lucru. Răspunsul este foarte simplu: la fiecare apel de funcţie membru se mai transmite un parametru suplimentar (invizibil utilizatorului), şi anume un pointer la obiectul în cauză. Acest pointer se numeşte "this", şi poate fi utilizat şi de către utilizator. Cel mai simplu exemplu de utilizare a lui "this" poate fi urmărit în exemplul de mai jos:

```
void stack::print_stack(void)
{
 struct _stiva *a; // var.tampon
 a= this->head; // pentru referirea variabilei membru se uti-
                // lizeaza operatorul "-> "; in acest exemplu
                // folosirea lui "this" este redundanta
 // ...
}
```

2.7. "Redefinirea" operatorilor

Termenul consacrat pentru redefinirea operatorilor este cel de "operator overloading". În cele ce urmează, vom analiza modul prin care se permite schimbarea sensului (redefinirea) operatorilor limbajului C. Utilizarea unei astfel de facilităţti duce la scrierea unor programe deosebit de inteligibile. După cum am mai arătat, faptul că se pot scrie intrucţiuni de genul

```
complex a,b,c,d,e;
...
e=(a+b)*(c-d);
```

în locul greoaielor

```
complex a,b,c,d,e;
...
```

```
e=inmulteste(aduna(a,b),scade(c,d));
```

este un avantaj care nu mai necesită alte comentarii.

Înainte de a trece la prezentarea exemplului propriu-zis, vom mai face o observaţie interesantă. Putem defini o clasă "MATRICE" care implementează operaţii cu matrici (adunări, scăderi, înmulţiri, transpuse, inverse etc.). Dacă facem acest lucru în mod "cinstit" e foarte bine. Dacă însă dorim să obţinem programe greu de descifrat, putem denumi operaţiile complet aleator. Operatorul "+" va efectua înmulţirile, "-" inversa, iar "*" adunarea. Dar atenţie: deşi acest lucru duce la scrierea unor programe nelizibile, însuşi programatorul are şanse mari să nu-şi mai poată ocoli proprile-i capcane.

În continuare vom analiza redefinirea operatorului de adunare de către o clasă "complex".

Exemplu:

```
# include <iostream.h>

class complex
{
 private:
    float real,imaginar;
 public:
    complex(int r=0,int i=1) { real=r;imaginar=i;}
    complex& operator+(complex&);
    void afiseaza_complex(void);
};

// creaza un nou numar complex care va fi initializat cu suma
// partilor reale, respectiv imaginare
complex& complex::operator+(complex& s2)
{
 complex& t= * new complex (real+s2.real,imaginar+
                            s2.imaginar);
// dupa cum am mai spus, in cadrul unei metode este permisa
// accesarea tuturor membrilor clasei; din acest motiv se pot
// face si accesari de genul "s2.real"
 return t;
}
// Tipareste partile reale si imaginare ale unui numar complex
void complex::afiseaza_complex(void)
{
 cout<<"\nComplex : "<<real<<"+i*"<<imaginar;
}
```



```
void main(void)
{
 complex a(10,11),b(1,1);
 complex& c=a+b;

 a.afiseaza_complex();
 b.afiseaza_complex();
 c.afiseaza_complex();
}
```

După execuţia programului, pe ecran se vor afişa mesajele:

```
Complex : 10+i*11
Complex : 1+i*1
Complex : 11+i*12
```

Analizînd corpul operatorului redefinit "+", remarcăm că el efectuează adunarea a două numere complexe. Sînt însumate succesiv părţile reale, respectiv imaginare. Să facem şi cîteva sublinieri:

- Pentru a putea fi returnat, rezultatului i se alocă dinamic memorie (dacă s-ar aloca static, obiectul ar fi distrus la părăsirea funcţiei).
- Apelul operatorului "+" poate avea două forme echivalente:

```
complex a,b,c;
...
a=b+c;
b.operator+(c);
```

  Se remarcă atît comoditatea, cît şi forţa sugestiei primei exprimări.

- La crearea dinamică a obiectului "complex" se va efectua şi apelul constructorului (cu parametri actuali specificaţi în listă).

2.8. FRIENDS

Exact ca în viaţă, şi în programare "prietenul la nevoie se cunoaşte". Dar despre ce este vorba? Uneori, se face simţită nevoia ca, în afara funcţiilor membru, şi alte funcţii să poată accesa varabilele private ale unei clase. Acest lucru se poate realiza prin mecanismul declarării respectivelor funcţii drept "friends" (prieteni) ai clasei respective. Astfel de declaraţii de funcţii se vor face întotdeauna în cadrul declarării clasei în cauză. În urma unei declaraţii de acest gen, "prietenii" au dreptul să acceseze datele private ale clasei respective. Facilitatea ca o funcţie să fie "prietenă" cu mai multe clase deschide un orizont foarte larg acestei idei. Iată şi un exemplu care rezolvă exact aceeaşi problemă a adunării numerelor complexe, folosind de astă dată o funcţie

"friend".

```
# include <iostream.h>

class complex
{
 private:
     float real,imaginar;
 public:
     complex(int r=0,int i=1) { real=r;imaginar=i;}
     friend complex& operator+(complex&,complex&);
     void afiseaza_complex(void);
};

complex& operator+(complex& s1,complex& s2)
{
 complex& t=*new complex(s1.real+s2.real,
                         s1.imaginar+s2.imaginar);
 return t;
}

void complex::afiseaza_complex(void)
{
 cout<<"\nComplex : "<<real<<"+i*"<<imaginar;
}

void main(void)
{
 complex a(10,11),b(1,1);
 complex& c=a+b;

 a.afiseaza_complex();
 b.afiseaza_complex();
 c.afiseaza_complex();
}
```

După execuţia programului, pe ecran se va afişa:

```
Complex : 10+i*11
Complex : 1+i*1
Complex : 11+i*12
```

În programul de mai sus, "prietenul" nu e o funcţie oarecare, ci chiar un operator redefinit. Ca şi în secţiunea "2.7", este vorba tot de un operator binar. Diferenţa faţă de secţiunea precedentă constă în faptul că, în acel

exemplu, binaritatea operatorului era oarecum mascată de către parametrul ascuns "this", pe cînd acum ea este cum nu se poate mai evidentă.

Ca ultim sfat, nu uitaţi cealaltă cunoscută zicală românească: "Fereşte-mă doamne de prieteni, că de duşmani mă feresc eu şi singur!". Este cum nu se poate mai adevărat, mai ales că funcţiile "friend" crează o breşă considerabilă în conceptul de încapsulare.

2.9. Clase derivate

Un concept important al programării OOP este cel de MOŞTENIRE. La baza lui stă un principiu foarte simplu: avînd o clasă oarecare "B" şi dorind să obţinem o altă clasă "D", asemănătoare cu prima, putem să declarăm pe "D" ca fiind DERIVATĂ din "B". În acest mod, "D" va moşteni atît variabilele cît şi funcţiile membru ale lui "B", pe lîngă care i se pot adăuga şi altele noi. Prin definiţie, "B" se va numi CLASA DE BAZĂ, iar "D" va purta numele de CLASADERIVATĂ.

Să luăm spre exemplu clasa "stack", care implementează o stivă de stringuri. Modelarea se face cu ajutorul unor liste simplu înlănţuite. Practic, exemplul din secţiunea "2.6" este copiat identic, atît că nu într-un fişier, ci în două.

Primul dintre ele se numeşte "stack.h" şi conţine declararea clasei:

```
class stack {
 protected :
     struct _stiva {
         char info[20];
         struct _stiva *next;
     } *head;     // capul stivei de stringuri

 public:
     stack(void);
     ~stack(void);

     void pop(void);
     void push(char *);
     void print_stack(void);
};
```

Unica schimbare apărută în această zonă a programului este prefixarea declarării lui "head" cu "protected" în loc de "private". Scopul acestei modificări se va vedea în curînd.

Cel de al doilea fişier va purta numele "stack.cpp" şi va arăta astfel:

```
# include <iostream.h>
# include <process.h>
# include <string.h>
# include <stdio.h>

# include "stack.h"

stack::stack(void) {
 head=NULL;
}

stack::~stack(void) {
 struct _stiva *a;     // var. tampon

 if(head!=NULL)
    do
    {
        a=head->next;
        delete head;
        head=a;
    }
    while (a!=NULL);
}

void stack::print_stack(void)
{
 struct _stiva *a;     // var. tampon

 a= head;
 if(head!=NULL)  {
    cout<<"\n STIVA CONTINE :\n";
        do {
            cout<<a->info<<"\n";
            a=a->next;
        }
        while (a!=NULL);
    cout<<"\n";
  }
  else cout<<"\n STIVA E VIDA !";
}

void stack::pop(void)
{
```



```
 struct _stiva *a;

 if (head!=NULL) {
    a=head;
    head=head->next;
    delete a;
 }
 else cout<<"Nu se poate executa \"pop\" dintr-o
          stiva goala !";
}

void stack::push(char *info)
{
 struct _stiva *a=new struct _stiva;

 if (!a) {
    cout<<"Memorie insuficienta in STACK::PUSH\n";
    exit(1);
 }
 strcpy(a->info,info);
 a->next=head;
 head=a;
}
```

Această rescriere a exemplului anterior este necesară deoarece vom specifica un nou tip de stivă, care să permită pe lîngă operaţiile prezentate anterior şi operaţia de INDEXARE. Prin indexare se va înţelege accesarea directă a elementului "i" al stivei. În cazul în care pe stivă nu există "i" elemente, se va da un mesaj de avertizare.

Dacă am lucra în TURBO C neobiectual, fie am căuta vechiul fişier şi am încerca să-i aducem modificările de rigoare, fie am rescrie pur şi simplu totul. Din cîte vom vedea, limbajele obiectuale permit abordări mult mai elegante şi mai eficiente.

Vom începe prin a crea un nou fişier, pe nume "idx_stk.h" :

```
# include "stack.h"

class i_stack:public stack {
 int max;
 public:
 i_stack(void);
 void push(char *);
 void pop(void);
```

```
};
```

În noua clasă "i_stack", care moşteneşte clasa "stack". Ce înseamnă moştenirea în cazul nostru? Pe lîngă structura lui "stack", vom mai beneficia de o variabilă privată "max" (care va păstra numărul de elemente de pe stivă) şi de un nou operator "[]" (care va realiza accesarea indexată a stivei). În plus, clasa "i_stack" va redefini procedurile "pop" şi "push", şi va avea un nou constructor, "i_stack". Prefixul "public" existent în faţa lui "stack" specifică modul în care se moştenesc datele din clasa de bază "stack": toţi membrii vor rămîne în continuare "public" sau "protected", după cum au fost declaraţi în clasa de bază. Dacă "public" ar fi lipsit, informaţiile publice şi protejate ale clasei de bază deveneau private în cea derivată.

În acest moment este timpul să facem o precizare. Toate informaţiile declarate "private" în cadrul unei clase vor exista şi în clasa derivată, dar nu vor fi accesibile. Într-un fel, acest lucru este normal. În caz contrar ar însemna că însăşi ideea de dată privată - accesibilă doar funcţiilor membru ale clasei respective - este compromisă.

Unica diferenţă între membrii "private" şi "protected" constă în aceea că ultimii vor fi accesibili şi în eventualele clase derivate.

Să analizăm în continuare fişierul "idx_stk.cpp", în care sînt implementate noile proceduri ale lui "i_stack".

```
# include <iostream.h>
# include "idx_stk.h"

void i_stack::push(char *elem)
{
 max++;          // s-a mai adaugat un element pe stiva
 stack::push(elem); // apelul vechiului "push"
}

void i_stack::pop(void)
{
 max--;          // s-a eliminat un element de pe stiva
 stack::pop();   // apelul vechiului "pop"
}

 i_stack::i_stack(void)
{
// nu mai trebuie facuta atribuirea "head=NULL;" deoarece
// inainte de a se executa constructorul clasei derivate, se va
// apela IMPLICIT constructorul clasei de baza; acesta va
// efectua atribuirea respectiva
 max=0;
}
struct _stiva* i_stack::operator[](int index)
{
 if (index<max) {
    struct _stiva *temp=head; // variabila temporara
                              // folosita la parcurgerea
                              // stivei
    for(int i=0;i<index;i++) temp=temp->next;
    return temp;
 }
 else {
    cout<<"Indexul "<<index<<" depaseste
         dimensiunea indexului maxim al stivei ( "
        <<(max-1)<<" )";
    return 0;
 }
}
```

Trebuie să subliniem că noile proceduri "push" şi "pop" le apelează pe cele vechi de îndată ce şi-au îndeplinit obligaţiile suplimentare. Modul în care se face apelul (prefixarea cu "stack::") este obligatoriu dacă se doreşte evitarea unei recursivităţi infinite. În lipsa prefixului "stack::", compilatorul nu are de unde să ştie că e vorba de metoda clasei de bază, şi va efectua un nou apel al metodei clasei derivate. Cum condiţie de oprire nu există, apelurile recursive vor continua la infinit.

În final, să studiem ultimul fişier ce face parte din "PROJECT", şi anume "main.cpp":

```
# include <iostream.h>

# include "idx_stk.h"

void main(void)
{
 i_stack my_stack;

 my_stack.push("Aceasta");
 my_stack.push("este");
 my_stack.push("stiva");

 cout<<"\n\n\n"<<my_stack[0]->info<<"\n"<<
  my_stack[1]->info<<"\n"<<my_stack[2]->info<<"\n";
```

```
 my_stack.print_stack();

 my_stack.pop();
 my_stack.print_stack();

 cout<<"\n"<<my_stack[0]->info<<"\n"<<
 my_stack[1]->info<<"\n"<<my_stack[2]->info<<"\n";

// exprimarea "my_stack[2]" va da nastere unui avertisment de
// indice prea mare

 my_stack.pop();
 my_stack.pop();
 my_stack.pop();

 my_stack.print_stack();
 my_stack.push("Sfirsit");
 my_stack.print_stack();
}
```

Procedura "main" nu are alt scop decît să utilizeze noul tip de stivă în mai multe situaţii. Modul în care are loc execuţia programului reiese din analiza mesajelor afişate pe ecran:

```
stiva
este
Aceasta

  STIVA CONTINE :

stiva
este
Aceasta

  STIVA CONTINE :

este
Aceasta
Indexul 2 depaseste dimensiunea indexului maxim al stivei ( 1 )
este
Aceasta

Nu se poate executa "pop" dintr-o stiva goala !
```



```
  STIVA E VIDA !
  STIVA CONTINE :
Sfirsit
```

Poate vi s-ar părea normal că avertizarea privitoare la depăşirea de index să fie tiparită DUPĂ afişarea celor două elemente ale stivei. Lucrurile nu stau aşa deoarece tipărirea tuturor valorilor dintr-o înlănţuire de operatori "<<" nu are loc decît în urma evaluării fiecărei expresii. Cum încercarea de evaluare a lui "my_stack[2]->info" va provoca tipărirea mesajului de avertizare, atenţionarea va fi afişată ÎNAINTEA valorilor de pe stivă.

2.10. Polimorfism

Exemplul dat în secţiunea precedentă a evidenţiat un lucru interesant, şi anume că atît clasa de bază ("stack") cît şi cea derivată ("i_stack") posedă două metode (funcţii membru) avînd acelaşi nume: "push" şi "pop". În plus, fiecare dintre ele va fi apelată DOAR de obiectele de tipul corespunzător. Marele avantaj rezidă în faptul că redeclararea unei metode nu duce la pierderea posibilităţii de a o accesa pe cea iniţială. Toate aceste proprietăţi reunite poartă numele de polimorfism.

O altă problemă direct legată de polimorfism va reieţi din exemplul de mai jos. Fişierul care îl conţine se va numi "poli.cpp" şi va înlocui în PROJECT-ul din secţiunea "2.9" fişierul "main.cpp".

```
# include <iostream.h>

# include "idx_stk.h"

void main()
{
 stack     stack1;
 i_stack   stack2;

stack *   my_stack[2];

 my_stack[0]=&stack1;
 my_stack[1]=&stack2; // un pointer la o clasa de baza B poate
                      //  pastra pointeri la clase derivate din B
 // incarcarea primei stive
 my_stack[0]->push("Aceasta");
 my_stack[0]->push("este o");
 my_stack[0]->push("stiva simpla");
```

```
// incarcarea celei de a doua stive; de notat obligativitatea
// operatorului "cast", daca se doreste apelul metodei "push"
// a clasei derivate
 ((i_stack *) my_stack[1])->push("Aceasta");
 ((i_stack *) my_stack[1])->push("este o");
 ((i_stack *) my_stack[1])->push("stiva indexata");

 cout<<"Al doilea element al stivei indexate : "
      <<(* ((i_stack *) my_stack[1]))[1]->info;

 cout<<"\n\nPrima stiva :";
 my_stack[0]->print_stack();
 cout<<"A doua stiva :";
 my_stack[1]->print_stack();
}
```

După execuţia programului pe ecran vor fi afişate următoarele date:

Al doilea element al stivei indexate : este o

Prima stiva :

STIVA CONTINE :

stiva simpla
este o
Aceasta

A doua stiva :

STIVA CONTINE :

stiva indexata
este o
Aceasta

O primă concluzie trasă în urma studierii acestui nou exemplu: întotdeauna un pointer la o clasă de bază poate păstra un pointer la o clasă derivată din ea. Atît că, dacă nu păstrăm într-o manieră oarecare informaţiile referitoare la tipul pointer-ului, nu vom putea opera corect cu metodele disponibile. Din cîte se vede, atît în cazul în care se folosesc metode redefinite, cît şi în cel al metodelor noi apărute în cadrul tipului derivat, este nevoie ca asupra pointer-ului să se aplice operatorul "cast", care să-l transforme în pointer la entitatea reală. În schimb funcţiile rămase neschimbate în cadrul ambelor clase - la noi "print_stack" - pot fi utilizate fără "casting". Mai există o

categorie specială de metode - numite VIRTUALE - care sînt capabile să deosebească pointerii la clasa derivată de cei la clasa de bază. Graţie acestei proprietăţi, ele nu vor necesita efectuarea unui "casting", nici măcar în cazul apelării unor metode redefinite în clasele derivate. Dar despre aceste metode virtuale vom vorbi în capitolul 4.

Iată-ne la sfîrşitul părţii introductive. Pînă acum, am făcut o sumară trecere în revistă a noţiunilor şi conceptelor de bază ale limbajului C++. Ea ne-a permis să avem o vedere de ansamblu a ceea ce înseamnă de fapt acest limbaj. În capitolele ce urmează, vom analiza fiecare tehnică şi concept în parte, pornind de la o abordare mai generală şi ajungînd să cunoaştem cele mai ascunse secrete.

3. CLASE

3.1. Introducere

Numele purtat de una din versiunile anterioare limbajului C++ a fost "C with Classes". Acest nume se datora noutăţii introduse: conceptul de "clasă". Pe baza claselor s-au putut dezvolta, mai apoi, toate noile tehnici de lucru. Fără clase, moştenirea şi polimorfismul nu ar fi fost decît simple speculaţii teoretice.

Conceptul de clasă este novator deoarece permite implementarea unui mecanism de definire şi utilizare a noilor tipuri de date, descrise de către utilizator. Acest lucru se face de o manieră absolut identică cu cea a tipurilor de date predefinite.

În plus, clasele permit separarea detaliilor de implementare (mecanismul intern) de cele absolut necesare pentru utilizarea noului tip de date. O excelentă analogie s-ar putea face cu modul de construcţie a unui UNIT în TURBO PASCAL: utilizatorul nu are acces decît la funcţiile şi procedurile ce apar în cadrul secţiunii INTERFACE. Cele incluse doar în IMPLEMENTATION nu pot fi utilizate de către programator. Ele pot fi apelate numai de către funcţiile din secţiunea INTERFACE, în cadrul "corpurilor" lor. O altă comparaţie sugestivă ar fi cea cu bibliotecile din TURBO C. Programatorul poate apela orice funcţie de biblioteca dacă îi cunoaşte prototipul şi efectul. El nu are însă acces la codul sursă al respectivelor funcţii.

3.2. Declararea claselor

Vom începe prin a da o serie de trei exemple, din care va reieşi un prim avantaj al utilizării claselor.

Să presupunem că scriem un program care lucrează cu corpuri în trei dimensiuni. Dacă vom memora coordonatele unui punct în trei variabile X,Y,Z şi avem o procedură ce realizează o translaţie a punctului respectiv cu TX,TY,TZ putem scrie:

Exemplul 1

```
int X,Y,Z;

void translateaza(int TX,int TY,int TZ)
{
 X+=TX;
 Y+=TY;
 Z+=TZ;
}
```


Deşi o astfel de exprimare este corectă, ea nu reflectă nicidecum existenţa unei legături între cele trei variabile X,Y,Z. Nimic nu dă de înţeles că ele au o trăsătură comună: descriu poziţia unui punct în spaţiu. Mult mai sugestivă ar fi o exprimare de genul:

Exemplul 2

```
struct coordonate_3D{
 int X,Y,Z;
};

void translateaza(struct coordonate_3D punct,
                  int TX,int TY,int TZ)
{
 punct.X+=TX;
 punct.Y+=TY;
 punct.Z+=TZ;
}
```

Cu toate că este mult mai sugestiv, şi acest mod de abordare are un inconvenient. E vorba de faptul că legătura existentă între STRUCTURA definită şi FUNCŢIA de translaţie nu este pusă în evidenţă în nici un mod. Cu alte cuvinte, se simte nevoia unui formalism care să specifice clar ideea că funcţia "translateaza" nu poate fi apelată DECÎT pentru a opera asupra unor puncte. Această problemă este uşor de remediat în C++ prin folosirea claselor.

Exemplul 3

```
class coordonate_3D{   // declararea clasei "coordonate_3D"
private :
 int X,Y,Z;    // declarare variabile membru
public :
 void translateaza(int TX,int TY,
                   int TZ);// declarare functie membru
};

// definire functie membru
void coordonate_3D::translateaza(int TX,int TY,
                                 int TZ)
{
 X+=TX;
 Y+=TY;
```

```
  Z+=TZ;
}

// declararea a trei obiecte de tipul "coordonate_3D"
class coordonate_3D obiect1,obiect2,obiect3;
```

Pentru a putea continua expunerea, e necesar să dăm cîteva definiţii.

O CLASĂ – este un tip de date definit de către utilizator, dar care se comportă întocmai ca un tip predefinit.

O INSTANŢIERE – a unei clase "CLASA" (sau un OBIECT) este o variabilă declarată ca fiind de tipul "CLASA".

O VARIABILĂ MEMBRU – este o variabilă declarată în cadrul unei clase.

O METODĂ – este o funcţie declarată (sau definită) în cadrul unei clase, şi care poate accesa toate variabilele declarate în cadrul clasei. Metodele se mai numesc şi "FUNCŢII MEMBRU".

Prin MEMBRII unei clase vom înţelege totalitatea metodelor şi variabilelor membru ale clasei.

În exemplul de mai sus avem:

a. trei obiecte de tipul "coordonate_3D" (obiect1,obiect2,obiect3);
b. trei variabile membru (X,Y,Z);
c. o singură funcţie membru ("translateaza").

Să încercăm, deci, să vedem în ce constă o clasă. O sintaxă mult simplificată arată astfel:

```
class Nume_clasa {
 [ [ private : ]  Lista_membri_1 ]
 [ [ public  : ]  Lista_membri_2 ]
};
```

Cuvîntul cheie "class" indică faptul că urmează descrierea unei clase avînd numele "Nume_clasa" (în exemplul de mai sus: "coordonate_3D"). Ca o precizare suplimentară, numele clasei poate fi orice identificator, unic în cadrul domeniului de existenţă respectiv.

Descrierea propriu-zisă a clasei constă în cele două liste de membri, prefixate eventual de cuvintele cheie "private" şi "public".

În caz că listele de membri există, ele pot conţine orice tipuri de declaraţii şi definiţii de date şi funcţii. După cum am spus deja, toate aceste date şi

funcţii vor constitui MEMBRII clasei.

Cuvintele cheie "private" şi "public" împart clasa în două secţiuni: una privată şi una publică. Să vedem care este rolul fiecăreia dintre cele două secţiuni:

public: membrii aparţinînd acestei secţiuni pot fi accesaţi de către utilizator din orice punct al domeniului de existenţă al respectivei clase.

private: membrii aparţinînd acestei secţiuni (atît date cît şi funcţii) nu pot fi accesaţi DECÎT de către metodele clasei respective. Utlizatorul clasei nu va avea acces la ei decît prin intermediul metodelor publice.

Reluînd analogia cu conceptul de "UNIT" din Pascal, se pot echivala secţiunile "PRIVATE" cu "IMPLEMENTATION" şi "PUBLIC" cu "INTERFACE".

În acest moment, dispunem de suficiente informaţii pentru a ne da seama că, în C++, încapsularea are la bază două idei:

a. cu ajutorul claselor se vor lega structurile de date şi funcţiile destinate manipulăriilor;
b. secţiunile private şi publice fac posibilă separarea mecanismului intern de interfaţă cu utilizatorul.

O sintaxă mai rafinată a claselor - şi deci mai apropiată de cea reală - ar fi următoarea:

```
Specificator_clasa Nume_clasa {
 [ [ private : ] Lista_membri_1 ]
 [ [ public  : ] Lista_membri_2 ]
};
```

Din cîte se vede, unica diferenţă faţă de definiţia precedentă constă în înlocuirea lui "class" cu "Specificator_clasa". Acesta din urmă poate fi unul dintre următoarele cuvinte cheie: "class", "struct" sau "union". Ultimele două descriu structuri de date avînd aceleaşi proprietăţi ca şi în limbajul C neobiectual, cu două modificări:

a. li se pot ataşa funcţii membru;
b. pot fi compuse din două secţiuni (una publică şi una privată).

DIFERENŢA între "class" şi "struct" constă în faptul că, dacă lipsesc specificatorii "public" şi "private", toţi membrii lui "class" vor fi IMPLICIT privaţi, iar cei ai lui "struct" publici. Şi în cazul specificatorului "union", implicit, membrii vor fi tot de tip public. Aceste observaţii lămuresc multe semne de întrebare şi dau o imagine clară a ceea ce înseamnă o clasă. Cele

spuse pînă acum vor fi subliniate de secvenţa de mai jos:

```
struct i1 {
 int a;
 double b;
 float c(void);
// a,b,c() sint IMPLICIT publici
};

union i2 {
 int a;
 double b;
 float c(void);
// a,b,c() sint IMPLICIT publici
};

class i3 {
 int a;
 double b;
 float c(void);
// a,b,c() sint IMPLICIT privati
};
```

3.3. Membrii unei clase. "Scope access operator"

Este cazul să revenim asupra celor trei exemple prezentate în secţiunea precedentă. Dacă aruncăm o privire de ansamblu, observăm că, în primul exemplu, procedura "translateaza" operează asupra variabilelor globale X, Y, Z. În al doilea exemplu, ea va opera asupra membrilor X, Y, Z ai unei structuri "coordonate_3D", transmisă ca parametru. Aceste două cazuri erau foarte simple, apelul procedurii avînd loc în manieră obişnuită a C-ului. Ultimul exemplu aduce un aspect nou: metoda "translateaza" nu poate "acţiona" independent, ci doar ca funcţie apelată de un obiect de tipul "coordonate_3D".

Din cîte se vede, nu mai este necesar ca, în cadrul funcţiei, să se precizeze "identitatea" lui X, Y şi Z. Ele vor fi considerate implicit variabile membru ale unui obiect "coordonate_3D", specificat explicit în momentul apelului. Deşi maniera exactă în care se face acest lucru o vom vedea în secţiunea următoare, dăm totuşi un exemplu care să clarifice problema. Apelul metodei "translateaza" se poate face astfel:

```
obiect1.translateaza(2,3,4);
```

În acest mod se precizează că obiectul "obiect1" apelează metoda "translateaza" cu parametrii actuali 2, 3 şi 4. Efectul acestui apel constă în adunarea

valorilor 2, 3 şi 4 la variabilele membru corespunzătoare ale obiectului "obiect1".

Să revenim la definirea lui "translateaza":

```
void coordonate_3D::translateaza(int TX,int TY,
                                 int TZ)
{
 // ...
}
```

Se remarcă imediat "prefixarea" numelui procedurii cu numele clasei urmat de "::". Acest din urmă simbol poartă numele englezesc de "scope resolution operator" sau "scope access operator", şi este utilizat în operaţii de modificare a domeniului de vizibilitate implicit. Un exemplu tipic de utilizare a sa ar fi următorul:

Exemplul 1

```
int i;

void f1(void)
{
 int i;
//...
 i++;      // se va incrementa "i"-ul local
//...
}

void f2(void)
{
 int i;
//...
 ::i++;      // se va incrementa "i"-ul global

// operatorul "::" a facilitat accesarea variabilei cu numele "i"
// aflata intr-un domeniu de vizibilitate mai inalt, si,
// in consecinta, mascata de catre variabila locala "i"
//...
}
```

Privind exemplele de mai sus, trebuie să mai facem o subliniere: în stînga lui "::" nu poate fi decît un nume de clasă sau nimic. În al doilea caz, variabila prefixată trebuie să fie globală!

Vom continua cu un sfat al lui Bjarne Stroustrup ([STRO], p.151): de cîte ori este util, se recomandă folosirea unor nume de funcţii din bibliotecile

standard ("read", "write", "open" ş.a.m.d.) drept nume de funcţii utilizator, urmînd ca primele să poată fi reutilizate în următoarea manieră:

```
//definirea functiei utilizator
int clasa::fopen(char * pNume="fisier.dat"
                 char * pAtr="rt")
{
// ...
// apelul functiei de biblioteca
 ::fopen(pNume,pAtr);
//...
}
```

În continuare, prezentăm un exemplu în care se va utiliza operatorul "scope access" ("scope resolution") pentru a opera în cadrul unei metode atît cu variabile membru, cît şi cu variabile globale.

Exemplul 2

```
int X1=10;

struct _clasa_ {
 int X1;
 void fctie(void);
} c1,c2;

void _clasa_::fctie(void)
{
 X1++;       // se va incrementa variabila membru
 ::X1++;     // se va incrementa variabila globala
}
```

Dacă avem o clasă "CLASA" şi un membru al ei numit "MEMBRU", o expresie de genul "CLASA::MEMBRU" se va numi "qualified name". Scopul lui "qualified name" este de a permite accesarea oricărui membru, chiar dacă este ascuns de o altă declaraţie de variabilă.

Exemplul 3

```
class ex {
 int contor;
public :
 void fct(void);
} ob1;

void ex::fct(void)
{
// variabila locala diferita de membrul "contor"
 int contor;
 ex::contor=100; // variabila-membru
 for(contor=0;contor<10;contor++)
//"contor"-ul lui "for" este cel local
    ex::contor+=2; // variabila-membru!!!
}
```

Încheind discuţia legată de "scope access operator", vom sublinia că prezenţa numelui clasei în faţa funcţiei membru este absolut necesară. În lipsa ei nu s-ar putea face distincţia între diversele metode purtînd nume identice şi aparţinînd unor clase diferite.

O altă motivaţie a prezenţei numelui clasei este legată de variabilele cu care se operează în cadrul funcţiei-membru. Dacă analizăm ultimul exemplu referitor la clasa "coordonate_3D", se remarcă faptul că se operează direct cu variabilele membru X, Y, Z, fără a mai da indicaţii suplimentare privind identitatea lor reală. Toate variabilele utilizate în cadrul unei funcţii-membru a clasei "CLASA" şi nedeclarate în cadrul ei, se presupun implicit a fi membri ai lui "CLASA". Dacă, în urma compilării, presupunerea se adevereşte, totul va fi în regulă. În caz contrar se încearcă identificarea lor ca variabile globale. Eşecul acestei identificări va provoca erorile de compilare corespunzătoare.

Distingerea variabilelor membru ale obiectelor din aceeaşi clasă se realizează graţie unui mecanism invizibil programatorului. El are la bază ideea de a furniza oricărei funcţii-membru un parametru ascuns, care să reprezinte un pointer la obiectul asupra căruia s-a aplicat metoda. Astfel, în Exemplul 3 al secţiunii precedente, pe lîngă cei trei parametrii declaraţi TX, TY, TZ, la fiecare apel de genul:

```
p1.translateaza(10,-2,7);
```

se mai transmite şi un pointer la obiectul "p1" (care apelează metoda). În acest fel expresiile de tipul

```
X+=TX;
```

devin

```
this->X+=TX;
```

unde, prin "this", în C++, se desemnează pointer-ul transmis ca parametru ascuns. Acest cuvînt cheie "this" este un nume de variabilă (locală funcţiilor membru), accesibilă programatorului, conţinînd un pointer la obiectul care a apelat metoda.

De fapt, rolul lui "this" nu se mărgineşte la identificarea membrilor unui obiect. După cum vom vedea în capitolele ce urmează, există situaţii de

implementare în care utilizarea lui "this" este salvatoare. Să considerăm următorul exemplu: intenţionăm să creăm o clasă "listă dublu înlănţuită". Dacă dorim să asigurăm inserarea unui element în capul listei, clasa poate arăta astfel:

Exemplul 4

```
# define NULL 0

class lista {
     int numar;
     lista * urm,* prec;
     static lista* capLista;
 public :
     void init(int i) {
         numar=i;
     }

// inserare in capul listei dublu inlantuite
 void insert(void){
     urm=capLista;
     prec=NULL;
     if(capLista) capLista->prec=this;
     capLista=this;
 }
};

// pentru a se putea face initializarea de mai jos este necesara declararea
// lui "cap_lista" ca fiind "STATIC"; pentru alte amanunte despre membrii
// statici se va studia sectiunea "3.6"
lista * lista::capLista=NULL;

void main(void)
{
 lista l1;
 lista l2;
 lista l3;

 l1.init(1);     // obiectul "l1" apeleaza functia membru "init"
 l2.init(2);
 l3.init(3);

 l1.insert();    // obiectul "l1" va fi inserat in lista
 l2.insert();    // obiectul "l2" va fi inserat in lista
```



```
 l3.insert();    // obiectul "l3" va fi inserat in lista
}
```

Alte exemple importante de utilizare a lui "this" vom regăsi în secţiunea "3.10.".

Cu obiectele aparţinînd unei clase, se poate opera întocmai ca şi cu orice tipuri de date predefinite. În secvenţa de mai jos se vor da cîteva exemple de astfel de utilizări.

Exemplul 5

```
coordonate_3D p1,p2,p3; // declararea a 3 obiecte
coordonate_3D *ptr;     // declararea unui pointer la obiect

coordonate_3D sir[20];  // declararea unui sir de 20 de obiecte
coordonate_3D * formeaza(int,int,int);
coordonate_3D * aduna(coordonate_3D,coordonate_3D);
...
p1=p2;                  // atribuire corecta
coordonate_3D=p3;       // EROARE: operatie fara sens
ptr=sir;                // corect; "ptr" va pointa la "sir[0] "
```

Din acest ultim exemplu reiese o nouă facilitate C++. În urma definirii unei clase "B", nu mai este necesară prezenţa specificatorului clasei ("class", "struct" sau "union") la fiecare declarare a unui obiect de acest tip. Va fi suficientă specificarea numelui clasei. Există însă şi o excepţie de la această regulă:

Exemplul 6

```
class litere {
 // ...
};

class numere {
 // ...
};

// ATENTIE: aceasta functie NU ESTE metoda a clasei cu acelasi nume
int numere(void)
{
 // ...
}

void main(void)
{
```

```
litere l1,l2; // CORECT: prezenta specificatorului "class"
              // este optionala
numere n1;// EROARE: clasa "numere" se suprapune ca domeniu
          // domeniu de existenta cu functia ne-membru "numere"
class numere p1,p2;    // CORECT
int numar=numere();    // CORECT: este un apel de functie!
}
```

Cu alte cuvinte, domeniul de existenţă al unui nume de clasă începe în momentul declarării ei şi se termină la sfîrşitul blocului în care a fost declarată. Unicul caz mai ciudat este cel prezentat în Exemplul 6, cînd numele clasei este identic cu cel al unui alt identificator (în cazul nostru, un nume de funcţie). În astfel de situaţii, e necesară prefixarea lui "numere" cu "class", pentru a se putea face deosebirea între clasa şi funcţia cu acelaşi nume.

Urmărind exemplul de mai jos, vom descoperi o altă situaţie specială.

Exemplul 7

```
struct nr_complex;    // DECLARATIE INCOMPLETA
nr_complex * adunare(nr_complex n1,nr_comlex n2);
//...

struct nr_complex     // DECLARATIE COMPLETA
{
 // ...
};
```

Din cîte se vede, prin DECLARAŢIE INCOMPLETĂ se înţelege declararea unei clase FĂRĂ a i se specifica lista de membri. O DECLARAŢIE COMPLETĂ a unei clase va conţine şi lista membrilor ei. Atenţie: la declararea completă a unei clase trebuie specificaţi TOŢI membrii ei. Adăugările ulterioare nu sînt posibile. Declarările incomplete sînt deosebit de utile cînd se doreşte referirea unor clase încă nedefinite.

Şi încă o observaţie, care, chiar dacă pare banală, este foarte importantă. Toate metodele unei clase vor avea exact acelaşi domeniu de existenţă ("scope") ca şi clasa căreia îi aparţin.

3.4. Accesarea membrilor unei clase.Pointeri la metode

În limbajul C, utilizarea operatorilor nu are nici un sens dacă nu li se asociază şi variabile asupra cărora să opereze.


```
int i=0,j=1,k=0;
i++;        // corect
--j;        // corect
++(--);     // EROARE: nu are nici un sens,
            // operatorii nu au operand
```

Analog, în C++, la fiecare apel de funcţie membru trebuie specificat obiectul care o apelează. De asemenea, la fiecare utilizare de variabilă membru trebuie specificat obiectul căruia îi aparţine.

Exemplul 1

```
struct cl1 {
 int a,b;
 void init(int,int);
 void inc(void);
} obiect_1,* pobiect_2;

void cl1::init(int a1,int b1)
{
 a=a1;
 b=b1;
}

void cl1::inc(void)
{
 a++;
 b++;
}

void main(void)
{
// obiectului pointat de "p_obiect_2" i se va aloca dinamic memorie
 pobiect_2=new cl1;

// membrii lui "obiect_1" vor fi initializati
 obiect_1.init(10,11);

// membrii lui "* p_obiect_2" vor fi initializati
 pobiect_2->init(10,11);

// membrii lui "obiect_1" vor fi incrementati
 obiect_1.inc();
```

```
// se va actualiza valoarea membrului "a" obiectului "obiect_1"
 obiect_1.a=pobiect_2->a+obiect_1.b;
}
```

Din cîte se vede, modul de accesare a variabilelor membru publice se face folosind operatorii "." sau "->", după cum obiectul de care aparţin este desemnat prin nume sau printr-un pointer.

Variabilele membru private nu pot fi accesate decît în cadrul metodelor clasei respective. Exemplul de mai jos va lămuri modul în care se realizează acest lucru.

Exemplul 2

```
class exemplu
{
     int numar;
 public :
     void init(int nr=0);
     void aduna(exemplu& obiect,exemplu* pobiect);
};

// Metoda "init" initializeaza membrul "numar" la valoarea
// parametrului "nr"
void exemplu::init(int nr)
{
 numar=nr;
}

// Metoda "aduna" actualizeaza valoarea membrului "numar"
void exemplu::aduna(exemplu& obiect,
                    exemplu* pobiect)
{
 numar=numar+obiect.numar+pobiect->numar;
// Membrul "numar" al obiectului ce a apelat metoda poate fi accesat di-
// rect, fara nici o prefixare. In cadrul metodei mai pot fi accesate si vari-
// abile membru private ale altor obiecte de acelasi tip, cu unica menti-
// une ca trebuie precizat si obiectul in cauza. Specificarea acestuia se va
// face printr-un nume sau printr-un pointer.
}

void main(void)
{
 exemplu obiect1,obiect2,obiect3;

 obiect1.init(2);
```



```
 obiect2.init(3);
 obiect3.init(4);

 obiect1.numar=6; // EROARE: "numar" este privat, deci
                  //  inaccesibil in functii ne-membru
 obiect1.aduna(obiect2,&obiect3); // "obiect1" apeleaza
                                  //  metoda aduna

}
```

Sintaxa apelului unei funcţii membru poate avea două forme clasice:

```
nume_obiect.nume_metoda(lista_parametri);

nume_pointer_la_obiect->nume_metoda
     (lista_parametri);
```

O a treia metodă de apelare a funcţiilor membru este cea folosind "pointerii la metode". Este vorba de un concept relativ nou (chiar şi în scurta istorie a C++), care nu apăruse în prima versiune AT&T a limbajului. Compilatorul firmei BORLAND ia însă în considerare îmbunătăţirea adusă de Stroustrup în versiunile următoare. Iată-ne, deci, faţă-n faţă cu o metodă elegantă şi eficace de manipulare a funcţiilor membru cu ajutorul pointerilor. Dacă precizăm că definirea unui "pointer la o funcţie membru a clasei B" se face prin "B::*", exemplul de mai jos nu mai are nevoie de explicaţii.

Exemplul 3

```
class clasa {
     int contor;
 public:
     void init(int nr=0) { contor=nr; }
     int increment(void) { return contor++; }
};

// tipul "pointerLaMetoda" este un pointer la o metoda a clasei
// "clasa", metoda ce nu are parametri si care returneaza "int"

typedef int (clasa::* pointerLaMetoda)(void);

void main(void)
{
 clasa c1,* pc1=&c1;
 pointerLaMetoda pM=clasa::increment;

 c1.init(1);
 pc1->init(2);
```

```
 int i=(c1.*pM)();   // operatorii ".*" si "-> *" sint utilizati la
 i=(pc1->*pM)();     // manipularea pointerilor la metode
}
```

Noul tip de dată ("pointer la metodă") desemnează adresa unei funcţii membru "M". Reamintim că metodele unei clase au IMPLICIT un parametru tip "pointer la obiect". El se va transmite "ascuns", şi va reprezenta acel "this" existent în cadrul oricărei metode. Din acest motiv, parametrul în cauză nu va mai apare în lista de parametri ai funcţiei desemnate de pointer.

Acum, avînd informaţii suficiente despre conceptul de clasă, să vedem cum ar trebui să arate o clasă ideală, în concepţia lui Bjarne Stroustrup ([STRO], p.142):

a. să aibă un set restrîns de metode publice;
b. să poată fi considerată drept o "cutie neagră", care în loc de borne de intrare şi ieşire are un set de metode publice;
c. să permită modificarea structurii datelor fără a schimba şi interfaţa (aşa-zisele borne de intrare şi ieşire din cutia neagră).

3.5. Alte amănunte privind declararea claselor. Funcţii "inline"

Să încercăm acum să sintetizăm cele spuse pînă în acest moment printr-un exemplu.

Exemplul 1

```
# include <iostream.h>

class nr_rational {
private :
 int numitor , numarator;
 void tip(void) {
    cout<<"\nNumitor="<<numitor<<",numarator="
        <<numarator;
 }
public :
 double val;
 void init(int,int);
};

void nr_rational::init(int a,int b)
{
```



```
 numarator=a;
 numitor= b!=0 ? b:1;      // se evita impartirile cu 0
 val=numarator/numitor;
 tip();
}

void main(void)
{
 nr_rational r1;
 r1.init(17,3); // corect
 r1.tip();      // EROARE: functie inaccesibila utilizatorului
 r1.val++;      // corect
 r1.numitor--;  // EROARE: variabila membru privata
}
```

Din acest exemplu reies clar proprietăţile secţiunilor private şi publice. Utilitatea lor este deosebită în cazul în care o lucrare mare este abordată modular, fiecare modul fiind tratat de către o altă persoană. Astfel, odată pusă la punct structura interfeţelor (secţiunile "public"), fiecare programator îşi poate alege sau schimba forma de reprezentare a datelor din cadrul unei clase, fără ca acest lucru să-i afecteze pe ceilalţi (deoarece clasele vor fi utilizate DOAR prin intermediul secţiunilor publice corespunzătoare).

Specificatorii "private" şi "public" pot să apară în cadrul declarării claselor de oricîte ori, şi în orice ordine. Domeniul de existenţă al acestor cuvinte cheie este din momentul întîlnirii lor, pînă în cel al apariţiei următorului specificator sau al terminării declaraţiei clasei.

Exemplu 2

```
class ex {
     int i;     // implicit "i" este privat
 public :
     int j;
     float j1; // explicit "j" si "j1" sint publice
 private :
     int k;
     float k1; // explicit "k" si "k1" sint private
 public :
     int l;     // explicit "l" este public
};
```

Există o deosebire fundamentală între "class" şi "struct" pe de o parte, şi "union" pe de altă parte. Instanţierile primelor două conţin în fiecare moment valorile TUTUROR variabilelor din lista de membri. În schimb, obiectele de

tip "union" vor păstra la un moment dat valoarea UNUI SINGUR membru din lista specificată.

Aceia dintre dumneavoastră care au folosit intensiv "union"-uri în limbajul "C" ştiu prea bine ce greoi era mecanismul de setare şi accesare a unui "union". Era nevoie de un "struct" care să conţină un "union" şi un cîmp de identificare a valorii actualmente păstrate în acesta. În C++, totul poate fi înlocuit foarte elegant de către o clasă în care:

a. mai mulţi constructori (vezi secţiunea "3.7.") se ocupă de iniţializarea diverselor cîmpuri;
b. un set de funcţii de accesare a diverselor tipuri de valori va verifica în prealabil corectitudinea operaţiei.

Ar putea să vi se pară că totul este ca şi în C, atît că, în loc să fie prezentat oarecum dezordonat, este încapsulat într-o clasă. Nu aveţi dreptate decît parţial. În C++ beneficiem în plus de o interfaţă elegantă (metodele publice), care ascunde toate detaliile şi eforturile de implementare ale mecanismului (ceea ce nu e puţin lucru).

În acest moment începem să avem o imagine destul de clară despre ceea ce înseamnă unificarea conceptelor de date şi cod. Într-un program C++ scris corect (DIN PUNCT DE VEDERE OBIECTUAL), nu au voie să existe date fără cod sau invers. Totul va fi încapsulat în cadrul claselor. Utilizarea variabilelor globale sau a funcţiilor ne-membru va însemna, în cele mai multe din cazuri, fie o concepţie greşită a programului, fie o utilizare ineficientă a programării orientate obiect. Conform manualului de utilizare al TURBO C++ ([BOR1], p.130), "datele direcţionează fluxul codului, iar acesta din urmă modelează forma şi valoarea datelor".

Şi încă o observaţie: deoarece folosirea variabilelor membru publice distruge ideea de ascundere a datelor, se spune că este bine să se evite utilizarea lor. Cei care se supun acestei restricţii implementează un set de metode care se ocupă doar cu setarea şi accesarea valorii datelor private.

Exemplul 3

```
class ex {
     int i;
 public :
     int  get_i(void) { return i; }
     void set_i(int a) { i=a;}
};
```

Putem face o scurtă paranteză, în care să arătăm că unul din dezavantajele invocate de adversarii C-ului era slaba lui tipizare. Acest lucru a dus la apariţia ANSI C, care înlătură slăbiciunea versiunii clasice. Totuşi, marea diferenţă

între ANSI C şi C++ constă în aceea că primul nu este capabil decît să semnalizeze eventualele incompatibilităţi ce apar. În C++ însă, graţie metodelor de setare şi accesare a variabilelor private, acest gen de inconveniente nu numai că este îndepărtat, dar putem spune chiar că este tratat de o manieră calitativ superioară.

O altă chestiune este legată de problema încuibăririi claselor. Dacă urmărim exemplul de mai jos, vom vedea că astfel de încuibăriri sînt permise în limbajul C++.

Exemplul 4

```
class dreptunghi {
    struct punct {
        int x,y;
        void init_punct(int x1=0,int y1=0);
    } stinga_sus,dreapta_jos;
 public :
    void init_dreptunghi(int x1,int y1,int x2,
                         int y2)
    {
        stinga_sus.init_punct(x1,y1);
        dreapta_jos.init_punct(x2,y2);
    }
};

void punct::init_punct(int x1,int y1)
{
 x=x1;
 y=y1;
}

punct p1;   // CORECT: tipul "punct" este "vizibil" in tot fisierul
dreptunghi dreptunghi_1;

void main(void)
{
 dreptunghi_1.init_dreptunghi(10,11,12,13);
}
```

Este necesară o precizare deosebit de importantă. Deşi ar părea normal ca tipul de dată "punct" să fie inaccesibil utilizatorului (fiind definit în secţiunea "private"), situaţia e de fapt cu totul alta. Regula spune astfel: definirea în cadrul unei clase B a unui nou tip de date D nu va fi influenţată de secţiunea (publică sau privată) în care are loc această operaţie.

Se mai observă că, în consecinţă, la definirea metodei de iniţializare a clasei "punct", prefixarea numelui funcţiei se face doar cu numele clasei "punct", nu şi cu cel al clasei "dreptunghi".

Există însă o excepţie ce merită amintită. În cazul declarării unui tip enumerat în cadrul unei clase, acesta va respecta regulile de accesare a secţiunii în care a fost declarat. Cu alte cuvinte, o declarare a unui tip enumerat făcută în cadrul secţiunii private nu va putea fi utilizată DECÎT de către membrii respectivei clase.

Exemplul 5

```
class ex{
    enum t1 { a , b };
 public:
    enum t2 { c , d };
};

void main(void){
 int variabila;

 variabila=ex::a; // EROARE: "ex::a" nu e accesibil
 variabila=c;     // EROARE: "c" nu poate fi identificat decit
                  // daca e prefixat cu "ex::"

 variabila=ex::c; // CORECT
}
```

Exemplul de mai sus subliniază că valorile publice de tip enumerat nu vor putea fi utilizate DECÎT prefixate cu numele clasei în care au fost declarate.

Titlul acestei secţiuni conţinea şi termenul de "funcţii inline". Să vedem despre ce este vorba:

Exemplul 6

```
# include <iostream.h>

 class coordonate_3D{
 int X,Y,Z;      // variabile membru ale clasei

// Prin DEFINIREA metodei in interiorul declararii clasei,
// functia membru devine IMPLICIT "inline" !!!

 void translateaza(int TX,int TY,int TZ)
 {
    X+=TX;
```



```
    Y+=TY;
    Z+=TZ;
 }
 void tipareste(void); // DECLARAREA unei metode
};

// Prefixarea DEFINITIEI metodei cu cuvintul cheie "inline" este
// echivalenta cu definirea functiei membru in cadrul declaratiei clasei
inline void coordonate_3D::tipareste(void)
{
 cout<<"\n\tX="<<X<<"\tY="<<Y<<"\tZ="<<Z<<"\n";
}
```

Care este noutatea introdusă de acest exemplu? În loc să ne mulţumim să declarăm procedura "translateaza" în cadrul structurii "coordonate_3D", o vom şi defini tot în cadrul clasei.

Care sînt consecinţele faptului că procedura "translateaza" va fi de tip "inline"? Acest lucru înseamnă că, practic, ea nu va fi apelată, ci expandată la locul fiecărui apel. După cum am mai spus si in capitolul 2, este vorba de un mecanism elegant de definire a unor macrouri de tipul "# define".

Definirea unei funcţii în cadrul unei clase se mai numeşte declarare "inline" implicită. Există şi o declarare explicită, care se face ca în cazul metodei "tipareste": funcţia membru e doar declarată în cadrul clasei, iar în locul în care e definită va fi prefixată de cuvîntul cheie "inline". De subliniat că orice funcţie, indiferent că este sau nu metodă, poate fi declarată inline. Atît că procedurile ce nu sînt membru a unei clase nu pot fi declarate "inline" DECÎT EXPLICIT.

Observaţii:

a. Expandarea funcţiilor "inline" va fi făcută de aşa manieră încît domeniul de existenţă al variabilelor să nu fie nicidecum afectat.

b. Faţă de cazul în care se folosesc directivele "#define", la utilizarea mecanismului "inline" se va executa şi verificarea tipului parametrilor!

c. Nu întotdeauna este posibil ca o funcţie să fie "inline". Declararea implicită sau explicită a acestei opţiuni este necesară, dar nu şi suficientă pentru ca ea să devină realitate. Este exact situaţia declarării variabilelor de tip "register": dacă nu mai sînt regiştrii disponibili, această opţiune rămîne pur teoretică.

d. ESTE INTERZISĂ FOLOSIREA în cadrul funcţiilor "inline" a STRUCTURILOR REPETITIVE DE CONTROL ("for", "while", "do while").

Vă veţi întreba, poate, la ce sînt bune funcţiile "inline"? Ele sînt deosebit de eficiente în cazurile în care transmiterea parametrilor prin stivă (operaţie

lentă) este mai costisitoare (în timp de execuţie) decît efectuarea operaţiilor din corpul funcţiei.

Facem o paranteză pentru a vă aminti că utilizatorii limbajului de asamblare care ezită între folosirea macro-urilor şi cea a procedurilor, sînt ajutaţi în luarea deciziilor de o formulă matematică destul de simplă. Astfel, dacă notăm cu "X" numărul de instrucţiuni din rutină şi cu "C" numărul de apelări a ei, putem calcula

$$F = X * C / (X + 4 * C).$$

Decizia optimă se va lua astfel: rutina se va expanda ca macrou sau nu, după cum F este mai mic sau mai mare decît 1.

Analog, firma BORLAND face cîteva recomandări, în funcţie de care să se decidă utilizarea sau nu a mecanismului "inline". Astfel, o funcţie merită să fie "inline" dacă vor fi îndeplinite simultan condiţiile:

a. corpul procedurii trebuie să conţină cel mult trei instrucţiuni;

b. numărul parametrilor şi dimensiunea lor trebuie să fie semnificativă.

Ultima recomandare a BORLAND-ului ([BOR1], p.140) sună cam aşa: utilizarea funcţiilor "inline" este eficientă doar în cazul în care codul generat de compilator pentru corpul funcţiei este mai scurt decît cel necesar apelului ei. Avînd în vedere că o astfel de comparare e imposibil de estimat printr-o simplă privire, ni se sugerează să încercăm ambele soluţii şi să o alegem pe cea optimă. Decizia optimă nu se poate lua decît în urma analizării codului obiect.

3.6. Membrii statici

Spre deosebire de "auto", "register" şi "extern", cuvîntul cheie "static" poate fi utilizat în prefixarea membrilor unei clase. Evident că, odată declarat "static", membrul în cauză va avea proprietăţi diferite de membrii obişnuiţi (ne-statici). Aceste proprietăţi apar datorită faptului că membrii statici NU APARŢIN UNUI OBIECT ANUME, ci sînt COMUNI tuturor instanţierilor unei clase.

Exemplul 1

```
class exemplu {
 int i;
 // ...
 public:

// variabila membru "contor" este declarata "public" doar in scop didactic
// (pentru a fi accesibila in functia "main")
 static int contor;      // variabila membru statica

 void inc(void) { i++; }
 void inc_contor(void) { contor++; }
 void init(void) { i=0; }
// ...

 static void functie(exemplu *); // metoda statica
} ob1,ob2,ob3;

int exemplu::contor=0; // initializarea variabilei statice

void exemplu::functie(exemplu *ptrEx)
{
 i+=76; // EROARE: nu se cunoaste obiectul ce-l "poseda" pe "i"
 ptrEx->i++; // CORECT: este specificat "proprietarul" lui "i"
 contor++;   // CORECT: "contor" este o variabila membru
             // statica, ea nu apartine unui obiect anume
}

void main(void)
{
 ob1.init();
 ob2.init();
 ob3.init();

 ob1.inc(); // ob1.i=1;
 ob2.inc(); // ob2.i=1;
 ob3.inc(); // ob3.i=1;

 ob1.functie(&ob1);       // CORECT
 exemplu::functie(&ob2); // CORECT

 functie(); // INCORECT (in afara de cazul ca exista o
            // procedura ne-membru cu acest nume)

 ob1.inc_contor();  // contor=1;
 ob2.inc_contor();  // contor=2;
 ob3.inc_contor();  // contor=3;

 exemplu::contor+=6; // contor=9;
}
```

Din exemplul de mai sus se poate vedea modul de declarare al membrilor statici, precum şi cel de iniţializare a variabilelor membru statice. Pentru fiecare instanţiere a clasei "exemplu" (ob1, ob2, ob3) va exista CÎTE O variabilă "i" DISTINCTĂ şi O SINGURĂ variabilă "contor", COMUNĂ tuturor celor trei obiecte. Comportarea variabilei "contor" poate fi cel mai bine înţeleasă analizînd funcţia "main".

Pentru o variabilă-membru statică se rezervă o singură zonă de memoriei, COMUNĂ TUTUROR INSTANŢIERILOR CLASEI. Toate operaţiile ce se efectuează asupra ei au drept operand acea UNICĂ variabilă, COMUNĂtuturorinstanţierilor.

În ultima linie a lui "main" se vede un alt mod de accesare a variabilelor-membru statice. Deoarece ele nu aparţin unui obiect anume, pot fi prefixate doar de numele clasei, urmat de "scope access operator" ("::"). Deci, prefixarea variabilei membru statice cu "ob1." , "ob2." sau "ob3." nu este necesară. Ea poate fi însă folosită pentru a indica apartenenţa lui "contor" la clasa "exemplu". Întotdeauna, indiferent de prefixare, va fi incrementată aceeaşi unică va- riabilă "contor".

Din cîte se vede în exemplu, apelul metodelor statice se face exact ca şi accesarea variabilelor-membru statice. Deoarece metodele statice nu sînt apelate de un obiect anume, nu li se va transmite acel parametru ascuns "this". Deci, dacă într-o metodă statică dorim să operăm cu variabile-membru nestatice, vom fi obligaţi să furnizăm un parametru explicit de genul obiect, pointer sau referinţă la obiect.

O trăsătură specifică variabilelor-membru statice aparţinînd unor clase declarate globale este aceea că TREBUIE iniţializate în cadrul fişierului în care clasa a fost declarată.

Exemplul 2

```
class ex {
    static float a;  // DECLARARE
 public :
    static float b;  // DECLARARE
};

float ex::a=10;    // INITIALIZARE
float ex::b=12;    // INITIALIZARE
```

În cazul în care domeniul de existenţă al clasei este local, iniţializarea variabilelor-membru statice nu mai este permisă.

Toţi membrii statici sînt DOAR DECLARAŢI în cadrul clasei, ei urmînd a fi obligatoriu INIŢIALIZAŢI. În acest moment facem o observaţie

care ţine mai degrabă de istoricul produselor BORLAND decît de limbajul C++. În versiunile anterioare produsului BORLAND C++ 2.0, iniţializarea variabilelor membru statice nu era obligatorie, ca făcîndu-se IMPLICIT cu valoarea 0. La compilatoarele mai noi iniţializarea devine OBLIGATORIE, lipsa ei fiind semnalizată ca o eroare de compilare.

Principala utilizare a variabilelor-membru statice constă în eliminarea cît mai multor variabile globale ale unui program.

3.7. Constructori

3.7.1. Iniţializarea structurilor de date

Pentru a înţelege cît mai exact conceptul de "constructor" vom face o scurtă paranteză, legată de un domeniu foarte bine conturat: iniţializarea datelor. Este bine cunoscut că orice programator are grijă să nu utilizeze variabile neiniţializate. Legat de acest lucru, să vedem cum se poate face o iniţializare în C.

Exemplul 1

```
# include <string.h>

# define MAX 20

struct _adresa {
 char nume[MAX],prenume[MAX],strada[MAX];
 int numar;
} adr1={"Caragiale","Ion Luca","Pacientei",13};

void main(void)
{
 struct _adresa adr2;

 strcpy(adr2.nume,"Matei");
 strcpy(adr2.prenume,"Basarab");
 strcpy(adr2.strada,"Postei");
 adr2.numar=21;
}
```

Din acest exemplu reies foarte clar două moduri de a iniţializa structuri de date: fie în momentul declarării lor (cazul lui "adr1"), fie prin accesarea directă a diverselor cîmpuri ale variabilei ("adr2"). Cel puţin al doilea mod de lucru este deosebit de incomod, în sensul că o serie de instrucţiuni se cer repetate de un număr deranjant de mare de ori, pentru a efectua o aceeaşi succesiune

de operaţii. În plus, există riscul ca, din neatenţie, în procesul de iniţializare să fie omise unele cîmpuri.

Se poate invoca şi o altă obiecţie, mult mai serioasă: clasele care au variabile membru private nu admit iniţializări prin liste! Acest fapt este bine ilustrat de exemplul de mai jos.

Exemplul 2

```
struct s1 {
 int a;
 char c[10];
}e1={10,"abcdef"};      // CORECT

struct s2 {
 private :
    int a;
    double b;
} e2={1,2};      // EROARE

class s3 {
 int a;
 char c;
} e3={12,'c'};      // EROARE
```

De aceea, este preferabil ca iniţializările să se execute într-o procedură, aşa cum se vede în exempul următor:

Exemplul 3

```
# include <string.h>

# define MAX 20

struct _adresa {
 char nume[MAX],prenume[MAX],strada[MAX];
 int numar;
} adr1;

void init(struct _adresa* pa,char* nume,
        char* prenume,char* strada,int nr)
{
 strcpy(pa->nume,nume);
 strcpy(pa->prenume,prenume);
 strcpy(pa->strada,strada);
 pa->numar=nr;
}
```



```
void main(void)
{
 struct _adresa adr2;

 init(&adr1,"Ion","Marin","Pesterii",11);
 init(&adr2,"Bordoc","Ecaterina","Plaiului",22);
}
```

Dar şi în acest caz există o obiecţie: nu există nici o legătură implicită între procedura "init" şi structura "_adresa". Într-o abordare obiectuală, am putea elimina inconvenientul declarînd respectiva procedură ca fiind o metodă a clasei "_adresa". Cu toate acestea, s-ar face în continuare simţită lipsa unui mod elegant de iniţializare de genul:

```
int a=0,b=7;
float z=67.87;
```

Pentru a elimina aceste neajunsuri, a fost necesară implementarea unui nou mecanism, şi anume cel al constructorilor. Să începem cu un exemplu, după care vom încerca o definire riguroasă a conceptului:

Exemplul 4

```
# include <string.h>

# define MAX 20

class _adresa {
     char nume[MAX],prenume[MAX],strada[MAX];
     int numar;
 public :
     // CONSTRUCTOR!!!
     _adresa(char *,char *,char *,int);
};

_adresa::_adresa(char* nume1="",char* prenume1="",
                 char* strada1="",int nr=0)
{
 strcpy(nume,nume1);
 strcpy(prenume,prenume1);
 strcpy(strada,strada1);
 numar=nr;
}

void main(void)
{
```

```
    _adresa adr1("Bordoc","Ecaterina");
    _adresa adr2;   // daca se doreste apelul constructorului cu toti
                    // parametrii luind valori implicite, se pot omite
                    // chiar si parantezele
}
```

3.7.2. Proprietăţi specifice constructorilor

În exemplul anterior observăm că un constructor este o procedură care are acelaşi nume ca şi clasa căreia îi aparţine. Deoarece servesc la crearea şi iniţializarea obiectelor, constructorii vor fi apelaţi ori de cîte ori se crează noi instanţieri ale unei clase. Utilitatea constructorilor este evidentă cel puţin sub două aspecte:

a. constructorul asigură iniţializarea corectă a tuturor variabilelor membru ale unui obiect;
b. constructorul oferă o garanţie în plus că iniţializarea unui obiect se va efectua o singură dată.

Să notăm şi faptul că o clasă poate avea mai mulţi constructori. În principiu, aceştia se declară, se definesc şi se utilizează întocmai ca orice metodă uzuală. Totuşi, există cîteva sublinieri absolut necesare:

- Constructorii poartă numele clasei căreia îi aparţin.
- Constructorii nu pot returna valori. În plus, o convenţie face ca nici la declararea şi nici la definirea lor să nu poată fi specificat "void" ca tip returnat.
- Adresa constructorilor nu este accesibilă programatorului. Expresiile de genul "&X::X()" sînt ilegale.
- Constructorii sînt apelaţi IMPLICIT de cîte ori este nevoie.
- În caz că o clasă nu are nici un constructor declarat de către programator, compilatorul va genera implicit unul. Acesta va fi public, fără nici un parametru, şi va avea o listă vidă de instrucţiuni.
- O clasă al cărei constructor a fost generat implicit nu poate fi utilizată ca membru într-un "union".
- În cadrul constructorilor se pot utiliza operatorii "new" şi "delete".
- Constructorii pot avea parametrii luînd valori implicite.

NOTĂ: Toate aceste observaţii sintetizate mai sus sînt valabile şi pentru destructori. Aceştia din urmă vor fi analizaţi în secţiunea următoare ("3.8.").

Asemeni funcţiilor obişnuite, un constructor poate efectua orice fel de acţiuni. Tot cu titlu de observaţie amintim că modalităţile de obţinere a unui

constructor "inline" sînt identice cu cele de la funcţiile-membru obişnuite.

Trebuie să subliniem că, la orice creare dinamică a unui obiect, compilatorul va apela IMPLICIT constructorul potrivit.

Cu riscul de a ne repeta, reamintim că o clasă poate avea oricîţi constructori. Aceştia vor avea acelaşi nume, dar vor diferi prin tipul şi numărul parametrilor. Cu toate acestea, indiferent de modul în care se crează un obiect, întotdeauna compilatorul va apela constructorul corespunzător (din punct de vedere al potrivirii parametrilor).

Exemplul 1

```
# include <string.h>

# define MAX 20

struct _adresa {
 char nume[MAX],prenume[MAX],strada[MAX];
 int numar;
 _adresa(char *,char *,char *,int);
 _adresa(_adresa&);
} adr1("Ion","Marin","Pesterii",13);

_adresa::_adresa(char* nume1="",char* prenume1="",
                 char* strada1="",int nr=0)
{
 strcpy(nume,nume1);
 strcpy(prenume,prenume1);
 strcpy(strada,strada1);
 numar=nr;
}

_adresa::_adresa(_adresa& ar)
{
 strcpy(nume,ar.nume);
 strcpy(prenume,ar.prenume);
 strcpy(strada,ar.strada);
 numar=ar.numar;
}

void main(void)
{
 struct _adresa adr2("Bordoc","Ecaterina");
 struct _adresa adr3=adr1; // se apeleaza constructorul
```

```
                                    // "_adresa(adresa&)"

    adr3=adr2;      // se executa o copie bit cu bit a tuturor
                    // membrilor clasei "_adresa"
}
```

Acest exemplu îl reia pe precedentul, atît că îi mai adaugă un constructor. Noul constructor este utilizat la iniţializarea unui obiect (în cazul nostru "adr3") cu o valoare de acelaşi tip (respectiv "adr1"). Practic, instrucţiunea:

```
struct _adresa adr3=adr1;
```

va fi interpretată sub forma echivalentă

```
struct _adresa adr3(adr1);
```

La executarea instrucţiunii "adr3=adr2;" se va folosi un procedeu de copiere bit cu bit, care nu trebuie confundat cu constructorul mai sus amintit. Altfel spus, deşi cele două instrucţiuni par a se executa identic, în primul caz se va apela constructorul "adresa(adresa&)", pe cînd în al doilea nu. Acest lucru poate fi foarte bine urmărit cu ajutorul Debugger-ului încorporat, prin folosirea trasării pas cu pas. În primul caz se va intra în funcţia "_adresa::_adresa(_adresa&)", iar în al doilea nu.

Este esenţială reţinerea unei restricţii: un constructor nu poate avea ca parametru o instanţiere a clasei respective. Acesta este motivul pentru care noul constructor adăugat de noi are prototipul "_adresa(_adresa&)" şi nu "_adresa(_adresa)".

De notat că apelul constructorilor variabilelor globale se face înainte de execuţia funcţiei "main". O astfel de observaţie este deosebit de importantă la efectuarea unor operaţii care necesită anumite condiţii iniţiale. De exemplu, să presupunem ca în cadrul constructorului clasei "grafic" avem o funcţie din biblioteca grafică. Dacă modul grafic de lucru e iniţializat doar în "main", existenţa unei variabile globale de tip "grafic" va provoca o eroare de execuţie. Tot ca observaţie, amintim că apelul constructorilor variabilelor globale VA PRECEDA chiar şi pe cel al funcţiilor declarate a se executa înainte de "main" (printr-o directivă de tip "#pragma").

3.7.3. Tipuri de constructori. Ambiguităţi

În exemplul de mai jos, vom declara două tipuri de constructori:

```
class X {
// .....
 X(X&);      // COPY CONSTRUCTOR: constructor de copiere
 X(void);    // DEFAULT CONSTRUCTOR: constructor implicit
```



```
};
```

CONSTRUCTORUL IMPLICIT ("default constructor") poate fi obţinut în două moduri:

a. Prin definirea de către utilizator a unui constructor fără nici un parametru. În acest caz corpul constructorului poate conţine orice fel de instrucţiuni admise într-un context dat (în sensul că, de exemplu, dacă e vorba de un constructor "inline", el nu poate conţine instrucţiuni repetitive).
b. Prin generarea sa IMPLICITĂ de către compilator. Un astfel de constructor va fi creat ori de cîte ori programatorul declară o clasă care nu are nici un constructor. În acest caz corpul constructorului nu va conţine nici o instrucţiune.

Am crezut necesar să insistăm asupra constructorului implicit deoarece deseori au loc unele confuzii. Cîtă vreme prin acest termen va fi desemnat un constructor fără nici un parametru, indiferent că a fost creat de programator ori generat de compilator, totul va fi în regulă. În general, se greşeşte înţelegînd prin CONSTRUCTOR IMPLICIT doar una din cele două categorii amintite.

Un alt lucru deosebit de important este legat de faptul că orice clasă trebuie să posede un constructor de copiere! Pentru a creşte flexibilitatea limbajului, compilatorul va genera IMPLICIT un astfel de constructor pentru fiecare clasă la care programatorul nu a facut-o explicit. Constructorul de copiere generat implicit (a nu se confunda cu constructorul implicit) va copia membru-cu-membru toate variabilele argumentului în cele ale obiectului care apelează metoda. El va fi apelat în situaţii de genul

```
X obiect2=obiect1;
```

sub forma echivalentă

```
X obiect2(obiect1);
```

Se recomandă o mare prudenţă în îmbinarea prezenţei constructorului implicit cu cea a unor constructori cu parametrii luînd valori implicite. În caz contrar, compilatorul va semnaliza erori datorate unor ambiguităţi de genul celor din exemplul de mai jos.

Exemplul 1

```
class nr {
     int numar;
 public :
     nr(void);
     nr(int=0);
     nr(int,int=0);
```

```
};

nr::nr(void)
{
 // ...
}

nr::nr(int nn)
{
 // ...
}

nr::nr(int nn1,int nn2)
{
 // ...
}

void main()
{
 nr n0;       // AMBIGUU: constructor implicit (fara parametrii)
              // sau cu un parametru luind valoare implicita?

 nr n2(1);    // AMBIGUU: constructor cu un parametru sau cu
              // doi, din care unul luind valoare implicita?

  nr n3(1,2);  // corect
 }
```

Din acest exemplu, pe lîngă problema ambiguităţii, mai trebuie reţinut un aspect. Parametrii unei funcţii-membru pot fi declaraţi ca luînd valori implicite fie la declararea metodei, fie la definirea ei. Nu se acceptă însă specificarea acestui lucru în ambele locuri!

Revenind la problema iniţializării obiectelor, vom face o nouă observaţie. Instanţierile claselor ce au constructori ne-impliciţi pot fi iniţializate în două moduri:

- Cu constructori de copiere.
- Cu constructori apelaţi cu o listă de parametrii. În acest caz există două posibilităţi echivalente de specificare a acţiunii de iniţializare:

Exemplul 2

```
class complex {
 double p_r,p_i;
 public:
    complex(int r,int i) { p_r=r; p_i=i; }
};
```


Exemplul3

```
class complex {
    double p_r,p_i;
 public:
    complex(int r,int i) : p_r(r),p_i(i) { }
};
```

3.7.4.Conversii

În continuare, va fi studiată o situaţie aparte. Este vorba de un hibrid între cazurile indicate în observaţia de mai sus. Să considerăm exemplul care urmează.

Exemplul 1

```
# include <string.h>

# define MAX 20

struct persoana {
 char numele[MAX],prenumele[MAX];
};

struct _adresa {
 char nume[MAX],prenume[MAX],strada[MAX];
 int numar;
 _adresa(persoana&);
};

_adresa::_adresa(persoana& p)
{
 strcpy(nume,p.numele);
 strcpy(prenume,p.prenumele);
 strcpy(strada,"Ploiesti");
 numar=5;
}

void main(void)
{
 persoana pers1={"Balota","Ion"};
```

```
    _adresa adr2=pers1;
}
```

Se observă că, în cadrul funcţiei "main", se încearcă iniţializarea lui "adr2" cu ajutorul unui constructor de copiere. Atît că după semnul de atribuire nu urmează obiect de tipul "_adresa", ci unul de tipul "persoana". În astfel de cazuri nu se poate recurge la constructorul de copiere (deoarece TIPURILE NU COINCID). Din acest motiv se va încerca un apel de constructor cu lista de parametri, la care primul parametru să fie de tipul "persoana&", iar restul fie că nu există, fie că iau valori implicite. Dacă există un astfel de constructor, el va fi apelat. Această operaţie poartă numele de "CONVERSIE UTILIZATOR" între tipurile "persoana" şi "_adresa".

Exemplul 2

```
class complex {
     int p_r,p_i;
 public :

// se presupune ca operatia de adunare intre 2 numere
// complexe este implementata ca metoda a clasei "complex"

     complex(int i=0 ,int j=1){
        p_r=i;
        p_i=j;
     }
}

void functie(void)
{
 complex a,b(1),c(2,3);
 a=b+1;          // se interpreteaza ca "a= b+ complex(1);"
}
```

Iată deci încă un tip de conversie definită de utilizator: din întreg (tip predefinit) în complex (tip definit de utilizator). Alte amănunte despre conversii vor fi prezentate în capitolul 5.

3.7.5. Apelul explicit al constructorilor

Înainte de a analiza secvenţa de mai jos, se impune o observaţie legată de comentariile ce o însoţesc. Fiecare comentariu conţine una sau două etichete. Aceste etichete sînt numere care indică ordinea de parcurgere a instrucţiunilor în cazul unei trasări pas cu pas cu ajutorul Debugger-ului. Altfel spus, este vorba chiar de ordinea în care calculatorul va executa instrucţiunile

programului. Am recurs la această metodă fiindcă este deosebit de expresivă din punctul de vedere al urmăririi acţiunii constructorilor. Deoarece o să mai folosim astfel de notaţii şi în alte exemple, vom face o convenţie: prin "linia 12" vom înţelege linia ce are eticheta 12, nu a 12-a linie a programului!

Exemplul 1

```
struct elem {
 int nr;
};

class clasa {
     int numar;
     struct elem * pElem;
 public:
     clasa(int);
};

clasa::clasa(int param=0)     // 3 si 8
{
 pElem=new elem;              // 4 si 9
 numar=pElem->nr=param;       // 5 si 10
}                             // 6 si 11

void main(void)               // 1
{
 clasa * pObiect;
 pObiect=new clasa(5);        // 2
 clasa Obiect=clasa(9);       // 7
 delete pObiect;              // 12
}                             // 13
```

În acest program se vor crea două instanţieri ale clasei "obiect", pentru una dintre ele alocîndu-se dinamic memorie. În ambele cazuri se va apela şi constructorul clasei respective.

În urma analizei exemplului de mai sus se impun cîteva observaţii:

a. în cadrul constructorului se poate aloca dinamic memorie (linia 4 şi 9);
b. constructorul poate fi apelat explicit în două moduri (vezi liniile 2 şi 7);
c. memoria alocată pentru obiectele create dinamic (vezi linia 2) se va elibera cu "delete" (linia 12).

3.7.6. Reducerea numărului de constructori

Există situaţii în care o clasă posedă un număr excesiv de constructori. De cele mai multe ori, aceste probleme pot fi evitate prin utilizarea judicioasă a mecanismului parametrilor luînd valori implicite.

Exemplul 1

```
// VARIANTA 1
// NUMAR EXCESIV DE CONSTRUCTORI !!!!

class ora_exacta_1 {
    int ora,minut,secunda;
 public:
    ora_exacta_1(int o,int m,
                 int s){ora=o;minut=m;secunda=s;}
    ora_exacta_1(int o,int m){ora=o;minut=m;
                              secunda=0;}
    ora_exacta_1(int o){ora=o;minut=secunda=0;}
    ora_exacta_1(void){ora=minut=secunda=0;}
};

// VARIANTA 2
class ora_exacta_2 {
    int ora,minut,secunda;
 public:
    ora_exacta_2(int o=0,int m=0,int s=0) {
       ora=o;
       minut=m;
       secunda=s;
    }
};

void main()
{
 ora_exacta_2 o1,o2(10),o3(10,10),o4(10,10,10);
}
```

3.7.7. Constructori şi obiecte statice

În cadrul acestei secţiuni trebuie făcute două observaţii:

a. Este obligatoriu ca parametrii constructorilor corespunzînd unor obiecte statice să fie EXPRESII CONSTANTE !!!

Exemplul 1

```
class ex {
 //...
 public :
    ex(int,double);
};

int f1(int a,double b,const int c)
{
 static ex ex1(2*c,2.3*c/5);// CORECT: "c" e constant
 static ex ex2(2,b);   // INCORECT: "b" e variabila
 static ex ex3(a,b);   // INCORECT: "a" si "b" sint variabile
}
```

b. Fie un program în care e definită o funcţie "F1". Să presupunem că:
   1. în funcţia "F1" este declarat un obiect static "o1";
   2. în cursul unei execuţii a programului nu se va apela niciodată funcţia "F1" (deoarece, să zicem, nu am selectat niciodată una din opţiunile meniului principal).

Se pune atunci problema: va fi sau nu va fi apelat constructorul obiectului static declarat in "F1" ? Conform specialiştilor firmei BORLAND, nu există o regulă clară care să specifice acest lucru. Ceea ce se ştie cu certitudine este faptul că, în caz de efectuare a apelului, acesta va avea loc DUPĂ cel al constructorilor corespunzînd variabilelor globale.

3.7.8. Constructorii claselor încuibărite

În exemplul de mai jos vom găsi o situaţie în care o clasă are ca membrii instanţieri ale altor clase. O astfel de situaţie de încuibărire trebuie tratată cu atenţia cuvenită, deoarece iniţializările nu sînt permise în cadrul declarării claselor. Din acest motiv există o convenţie care impune apelul constructorilor obiectelor membru.

Să considerăm o clasă "CLASA" ce conţine un obiect încuibărit de tip "OI", care, la rîndul său, are un membru "M" de tip pointer. Presupunem că în cadrul constructorului clasei "OI" se alocă dinamic o zonă de memorie la care să pointeze "M". Dacă în cadrul constructorului clasei "CLASA" se încearcă iniţializarea zonei la care pointează "M", şi constructorul lui "OI" nu a fost apelat, pointer-ul "M" va avea o valoare aleatoare, iar iniţializarea va fi ineficientă. În plus, prin această iniţializare a unei zone oarecare, s-ar putea distruge informaţii semnificative din memoria calculatorului. Pentru a evita astfel de riscuri, limbajul C++ prevede un mecanism extrem de simplu şi eficient.

Înainte de a analiza exemplul de mai jos, să vedem în cîteva cuvinte despre ce este vorba. Au fost declarate trei clase: una păstrează coordonatele unui punct şi două care definesc tipuri de vectori (definiţi prin cîte două puncte semnificative: origine şi vîrf). În continuare, vom urmări modul în care se vor apela constructorii corespunzători celor două puncte semnificative din cadrul fiecărui vector.

Exemplul 1

```
class punct {

// un punct este caracterizat de o abscisa si de o ordonata
     int x,y;

 public:
// cei doi constructori vor diferi doar prin tipul argumentelor;
// ei vor seta valorile absciseisi ordonatei
     punct(float a,float b)
     {
        x=a;
        y=b;
     }
     punct(int a=0,int b=0)
     {
        x=a;
        y=b;
     }
};

class vector1 {
  // un vector este caracterizat de doua puncte: originea si virful
     punct origine, virf;

  // variabila ce va contoriza numarul de instantieri ale clasei "vector1"
     static int   numar_vectori;
 public :
     vector1(float,float);
     vector1(int,int,int,int);
};

class vector2 {
// originea si virful vectorului vor fi specificate de cei doi pointeri
// ("pOrigine"si"pVirf")
     punct * pOrigine, * pVirf;
 public :
```



```
 vector2(int,int,int,int);
};

int vector1::numar_vectori=0;//initializarea membruluista-
                             //tic privat "numar_vectori"

vector1::vector1(float x1,float y1)
  : origine(x1,y1)
// - "origine"este initializat prin apelul explicit al constructorului avind doi
//   parametrii "float";
// - "virf" este initializat prin apelul implicit al constructorului avind doi
//   parametrii "int", ambii luind valoarea implicita 0
{
 numar_vectori++; // se incrementeaza numarul de instantieri a
                  // clasei "vector1"
}

vector1::vector1(int x1,int y1,int x2,int y2)
: origine(x1,y1) , virf(x2,y2)
// ambele puncte sint initializate prin apelul explicit al constructorului
// corespunzator
{
 numar_vectori++; // se incrementeaza numarul de instantieri a
                  // clasei "vector1"
}

vector2::vector2(int x1,int y1,int x2,int y2)
{
// se aloca memorie pentru cele doua puncte si se apeleza constructorul
// corespunzator
 pOrigine=new punct(x1,y1);
 pVirf=new punct(x2,y2);
}

void main(void)
{
 vector1 v1_1(10,10),v1_2(0,0,1,1);
 vector2 v2(16,16,17,19);
}
```

O primă observaţie ar fi legată de modul de apelare a constructorului "punct". În cadrul clasei "vector2", apelul constructorului "punct" se va face IMPLICIT în momentul în care se va ajunge la fiecare din cele două "new"-uri. Deci, în acest caz, totul decurge conform aşteptărilor.

În cazul lui "vector1" situaţia este cu totul alta. În cadrul constructorului avînd 4 parametri, se remarcă modul de apelare al constructorilor pentru cei doi membri "punct". Forma este foarte asemănătoare cu cea a iniţializării membrilor ce nu sînt obiecte. Atît că, în cazul claselor încuibărite care nu au nici constructori impliciţi şi nici constructori la care toţi parametrii pot lua valori implicite, este OBLIGATORIE această formă de iniţializare. Aşa cum se vede şi în cazul constructorului "vector1(float,float)", dacă obiectul membru are un constructor la care toţi parametrii pot lua valori implicite, această formă de "iniţializare" nu mai este obligatorie, compilatorul fiind capabil să decidă apelul constructorului la care toţi parametrii iau valori implicite. Analog s-ar petrece lucrurile şi în cazul existenţei unui constructor implicit.

După ce am înţeles ceea ce se întîmplă, să încercăm să dăm o regulă generală. În cazul în care într-un program există o clasă "CLS" care are ca membrii obiecte ale unor alte clase "CLS0", "CLS1", ..., pot apărea următoarele situaţii:

a. Dacă clasele încuibărite în cadrul lui "CLS" nu au constructori, nu va exista nici o deosebire între constructorii clasei "CLS" şi un constructor de clasă ce nu are obiecte-membru încuibărite.
b. Dacă clasele încuibărite nu au un constructor implicit şi constructor la care toţi parametrii pot lua valori implicite, este obligatorie specificarea EXPLICITĂ a numelui fiecărui obiect urmat de lista parametrilor actuali.
c. Dacă clasele încuibărite au un constructor implicit sau unul cu TOŢI parametrii avînd valori implicite, se poate proceda fie ca în cazul "a." (situaţie în care se vor apela IMPLICIT constructorii respectivi), fie că în cazul "b." (se vor specifica explicit doar parametrii doriţi). De exemplu:

```
vector1::vector1(float x1,float y1)
: origine(x1,y1),virf(10)
//...
```

unde vîrf este apelat cu un singur parametru.

Cu titlu de observaţie, subliniem că ÎNTOTDEAUNA apelul constructorilor obiectelor încuibărite se va face ÎNAINTE de executarea constructorului obiectului "cuib". Ceea ce nu se poate preciza cu exactitate este ORDINEA de apelare a constructorilor obiectelor încuibărite. Astfel, nu se recomandă scrierea unor secvenţe de genul

```
vector1::vector1(int x1,int y1,int x2,int y2)
    : origine(x1++,y1--) , virf(x1+x2,y1+y2)
{
 // ....
}
```


deoarece nu se poate stabili cu certitudine care dintre cei doi constructori "punct" va fi apelat primul.

Ca o regulă, vom preciza că specificarea constructorilor claselor încuibărite se va face OBLIGATORIU în cadrul DEFINIŢIEI constructorului "cuib", nu al declaraţiei lui.

În concluzie, putem spune că întotdeauna se vor apela constructorii claselor încuibărite, indiferent dacă modalitatea de apel va fi implicită sau explicită. În cazul în care apelul nu e explicit şi nici nu se poate face implicit (de către compilator), va fi semnalizată o eroare de compilare.

3.8. Destructori

3.8.1. Definire şi caracteristici

Destructorul poate fi privit ca şi complement al constructorului. Spuneam că, în general, constructorii sînt folosiţi pentru a aloca memorie şi pentru a efectua anumite operaţii (de exemplu, incrementarea unui contor al numărului de instanţieri ale clasei). Destructorii se utilizează îndeobşte la eliberarea memoriei alocate de primii şi la efectuarea unor operaţii inverse celor amintite mai sus (de exemplu, decrementarea aceluiaşi contor). Un alt amănunt ce indică această complementaritate ţine de modul în care se denumesc destructorii: ei poartă numele clasei precedat de un caracter "~".

```
class B {
 // ...
 public :
     B(void); // constructor
     ~B(void); // destructor
};
```

Aşa cum am afirmat şi la începutul secţiunii "3.7.2.", toate observaţiile făcute în legătură cu constructorii rămîn valabile şi în cazul destructorilor.

Exemplul 1

```
# define NULL 0

struct s{
 int nr;
 struct s * next;
};

class B {
    int i;
```

```
    struct s * ps;
public:
    B(int);
    ~B(void);
};

B::B(int ii=0)                  // 3 si 7
{
 ps=new s; ps->next=NULL; i=ps->nr=ii; // 4 si 8
                                       // 5 si 9
}

B::~B(void)             // 11 si 14
{
 delete ps;             // 12 si 15
}                       // 13 si 16

void main(void)         // 1
{
 B * pb;
 B b=9;                 // 2
 pb=new B(3);           // 6
 delete pb;             // 10
}                       // 17
```

Să facem cîteva sublinieri pe marginea programului prezentat:

a. Destructorii se apelează implicit în două situaţii: la eliberarea cu "delete" a memoriei alocate dinamic pentru memorarea unor obiecte (linia 10) sau la părăsirea domeniului de existenţă al unei variabile (linia 17; variabila "b").

b. Acest al doilea caz merită o menţiune specială: dacă este vorba de variabile globale sau definite în "main", distrugerea lor se va face DUPĂ ultima instrucţiune din "main", dar ÎNAINTE de încheierea execuţiei programului.

c. În cazul obiectelor create dinamic, destructorul nu se va apela DECÎT dacă se încearcă eliberarea (cu "delete") zonei de memorie (linia 10).

d. Dacă avem clase încuibărite, ordinea de apelare a destructorilor este inversă celei de apelare a constructorilor respectivi.

e. Ordinea apelurilor de funcţii executate la sfîrşitul unui program TURBO C++ este următoarea:
   - funcţiile "atexit" (în ordinea inserării lor);
   - funcţiile "#pragma exit" (în ordinea codului priorităţii);
   - destructorii variabilelor globale.

   Deoarece sînt mai rar folosite, s-ar putea ca primele două categorii de

funcţii să nu vă fie familiare. În acest caz puteţi găsi informaţii amănunţite în orice manual de TURBO C.

f. Dacă în timpul execuţiei unui program se va apela funcţia "exit", nu vor fi apelaţi destructorii variabilelor locale, ci doar cei ai variabilelor globale. În caz de apel a funcţiei "abort" nu va fi apelat nici un destructor!

g. Avînd în vedere cele de mai sus, nu se recomandă utilizarea lui "exit" în cadrul destructorilor. În caz contrar se poate intra într-o recursivitate infinită.

h. Utilizatorul dispune de două moduri de a provoca apelul unui destructor:
   - prin specificarea numelui său;

```
class B {
    //...
    public: ~B();
};

void f(void) {
    B b;
    b.B::~B();// APEL DIRECT: e obligatoriu prefixul "B::"
}
```

   - prin folosirea lui "delete" (este o metodă indirectă; vezi linia 10).

3.8.2. Destructorii claselor încuibărite

În această subsecţiune se va încheia discuţia despre clasele încuibărite, începută în cadrul secţiunii "3.5" şi continuată apoi în subsecţiunea "3.7.8.". În problema apelului destructorilor claselor încuibărite se impun doar două observaţii, pe care le vom discuta pe exemplul ce urmează.

```
class baza {
    class incuib1 ob1;
    class incuib2 ob2;
    // ...
 public :
    // ...
    ~baza(void);
} ob0;
```

Care ar fi sublinierile ce se impun:

a. Apelul destructorilor obiectelor "ob1" şi "ob2" se va face doar DUPĂ execuţia destructorului "~baza".

b. În cazul în care se preferă ca în loc de obiecte încuibărite să se folosească

membrii de tipul "pointer la obiect" (să-i zicem încuibărire indirectă), va fi necesară atît alocarea dinamică, cît şi eliberarea EXPLICITĂ a memoriei necesare pentru obiectele la care vor pointa membrii în cauză. De obicei astfel de acţiuni au loc în cadrul constructorului, respectiv destructorului clasei "cuib". Dar despre acest caz particular vom discuta în secţiunea "3.10.".

3.8.3. Erori deosebite

Un prim caz care se cere studiat cu atenţie este cel din exemplul de mai jos:

Exemplul 1

```
class ex {
    double * pNumar;
 public :
    ex(double);
    ~ex();
};

ex::ex(double nr)
{
 pNumar=new double;
 *pNumar=nr;
}

ex::~ex(void)
{
 // ...
 delete pNumar;
}

void main(void)
{
 ex b1(10.66);
 ex b2=b1;              // 1
 ex b3(20.101);
 b1=b2=b3;              // 2
}
```

Liniile etichetate cu 1 şi 2 sînt generatoare de erori subtile şi greu de detectat. La baza generării lor stă mecanismul de copiere bit cu bit a obiectelor. Acesta poate fi utilizat în două situaţii: la atribuiri sau la apeluri ale constructorului implicit de copiere. Dar atenţie: eroarea nu apare la copiere, ci la apelul destructorilor, care vor fi siliţi să elibereze de mai multe ori una şi aceeaşi

zonă de memorie. Acest fapt se datorează copierii în membrii lui "b1" şi "b2" a adresei memorate în "b3". În astfel de situaţii comportarea programului poate deveni bizară, de obicei provocînd blocarea sistemului. Practic, cazul exemplului de mai sus este echivalent cu cel din secvenţa care urmează:

Exemplul 2

```
void f(void)
{
 int* pInt = new int;
 delete pInt;
 delete pInt; // GRESEALA: aceeasi zona de memorie e
              // eliberata de doua ori
}
```

3.9. Friends

În general, se consideră că există patru faze ale conceperii unui program: întocmirea caietului de sarcini, realizarea proiectului conceptual, implementarea practică şi testarea produsului final. Dacă prima fază este corect executată, va fi imposibil ca proiectarea conceptuală să nu dea naştere unor ierarhii de clase bine puse la punct. În cele mai multe cazuri, o astfel de nereuşită se datorează nepriceperii programatorului, nu inabordabilităţii obiectuale a problemei. Cu toate acestea, Bjarne Stroustrup s-a gîndit să ofere soluţii elegante pentru rezolvarea tuturor situaţiilor. În acest scop, el a creat mecanismul de "friend", care să permită abateri controlate de la ideea de protecţie a datelor prin încapsulare. Înainte de a intra în detalii, repetam un avertisment: în caz că sînteţi tentat să folosiţi acest mecanism, vă recomandăm să mai meditaţi încă o dată asupra modului în care aţi modularizat problema propusă spre rezolvare! Aveţi şanse mari fie să găsiţi un decupaj mai bun, fie să puteţi exprima "prietenia" printr-o relaţie de "moştenire" (vezi capitolul 4.). Dacă tot nu aţi izbutit, puteţi fi aproape sigur că încă nu stăpîniţi perfect conceptele OOP.

Mecanismul de "friend" porneşte de la imposibilitatea ca una şi aceeaşi metodă să fie membru a mai multor clase. Avînd în vedere acest lucru, putem examina cele două aspecte ale problemei:

a. **FUNCŢII FRIEND**

· Este vorba de funcţii care nu sînt metode ale unei clase, dar care au totuşi acces la membrii privaţi ai acesteia. Orice funcţie poate fi "friend" al unei clase, indiferent că e vorba de funcţiile ordinare, metodele altor clase sau operatori redefiniţi (vezi capitolul 5.).

Exemplul 1

```
class B {
    int a;
    int f(void);
    friend int f1(B&);
 public :
    friend int M::f2(B&,int);
}

int f1(B& rB)
{
 return rB.f();
}

int M::f2(B& rB,int j=0){
 if(rB.a>7) return j++;
      else return j--
}
```

Aşa cum se vede şi în exemplul de mai sus, nu are nici o importanţă dacă o funcţie este declarată "friend" în cadrul secţiunii "private" sau "public" a unei clase. Iată deci o excepţie: declaraţia făcută în cadrul secţiunii "private" este accesibilă "în exteriorul" clasei.

b. CLASE FRIEND

În cazul în care dorim ca toţi membrii unei clase "M" să aibă acces la partea privată a unei clase "B", în loc să enumerăm metodele lui "M" ca fiind "friends" ai lui "B", putem declara direct clasa "M" drept "friend" al lui "B". De regulă, aceste situaţii se pot evita declarîndu-l pe "M" ca fiind derivat din "B" (adică "îl moşteneşte"). Amănunte despre modul de realizare al unor astfel de derivări vom vedea în capitolul 4.

Exemplul 2

```
class M {
 // ...
};

class B {
 // ...
 friend class M;
};
```

Observaţie: relaţia de "friend" NU ESTE TRANZITIVĂ! Adică, dacă clasa B1 este "friend" al lui B2, şi B2 este "friend" al lui B3, NU înseamnă

că B1 este implicit "friend" al lui B3.

După cum reiese şi din Exemplul 1, funcţiile "friend" nu au parametrul ascuns "this" (deoarece nu sînt metode). Această carenţă se va suplini prin transmiterea unor parametrii obişnuiţi de tip pointer, obiect sau referinţă la obiect.

În încheierea secţiunii, dorim să vă prezentăm un exemplu în care folosirea funcţiilor "friend" este foarte utilă din punctul de vedere al diminuării timpului de execuţie.

Exemplul 3

```
class rational;    // declarare incompleta

class complex {
 double p_reala,p_imaginara;
 friend complex& ponderare(complex&,rational&);
public :
 complex(double r,double i) : p_reala(r),
                              p_imaginara(i) {}
 double get_real(void)     { return p_reala;}
 double get_imaginar(void) {return p_imaginara;}
};

class rational {
 int numarator,numitor;
 double val;
public :
 friend complex& ponderare(complex&,rational&);

 rational(int n1,int n2) : numarator(n1)
 {
    numitor= n2!=0 ? n2:1;
    val=((double)numarator)/numitor;
 }

 double get_valoare(void) { return val;}
};

// fiind "friend", "ponderare" are acces la membrii privati ai
// claselor "complex" si "rational"
complex& ponderare(complex& c,rational& r)
{
 complex* t=new complex(c.p_reala*r.val,
                        c.p_imaginara*r.val);
```

```
 return *t;
}

// nefiind "friend", "ponderare_ineficienta" nu are acces la
// membrii privati ai claselor "complex" si "rational"
complex& ponderare_ineficienta(complex& c,
                               rational& r)
{
 complex* t=new complex(c.get_real()
                  *r.get_valoare(),
                 c.get_imaginar()*r.get_valoare());
 return *t;
}

void main(void)
{
 complex  a(2,4),b(6,9);
 rational d(1,2),e(1,3);

 a=ponderare(a,d);
 b=ponderare(b,e);

 a=ponderare_ineficienta(a,d);
 b=ponderare_ineficienta(b,e);
}
```

În exemplul ales sînt definite două funcţii de ponderare a unui număr complex prin unul raţional. Ele fac practic acelaşi lucru, însă prin metode diferite. Funcţia "ponderare" fiind "friend" a celor două clase, le poate accesa membrii privaţi, deci este foarte rapidă. Cea de a doua, "ponderare_ineficienta", nu accesează membrii privaţi ai celor două clase DECÎT prin intermediul metodelor publice. Din acest motiv timpul de acces este mai lung, iar funcţia mai lentă. Totuşi, la unul sau două apeluri de acest gen, avantajul nu e deloc evident. Dacă am dori însă ponderarea unui şir de 100000 de numere complexe, lucrurile se vor schimba net. De aceea se recomandă ca, în aplicaţiile în timp real (deci la care orice fracţiune de secundă contează: conducere de procese industriale, recunoaşteri de imagini, etc), să se folosească calea cea mai rapidă, chiar dacă aceasta nu este pur obiectuală.

3.10. Comportarea constructorilor şi destructorilor în situaţii necesitînd alocare dinamică de memorie

3.10.1. Crearea şi distrugerea şirurilor de obiecte

Începem prin a reaminti că un CONSTRUCTOR IMPLICIT este un constructor cu lista de parametrii vidă.

O restricţie oarecum supărătoare a limbajului C++ ar fi următoarea: nu putem declara un tablou de obiecte DECÎT dacă clasa respectivă are un CONSTRUCTOR IMPLICIT. Această restricţie este necesară datorită imposibilităţii de a furniza (în cadrul declaraţiei şirului) argumentele necesare constructorilor. Pentru a nu da naştere la confuzii, s-a decis ca, la iniţializarea obiectelor unui şir, să NU poată fi folosit nici măcar un constructor cu toţi parametrii luînd valori implicite.

Deoarece clasa care va da tipul tabloului poate avea ORICÎŢI constructori în afara celui implicit, ea nu-şi va pierde generalitatea. Atit că, în cazul în care respectiva clasă nu posedă un constructor implicit, trebuie făcută una din următoarele modificări:

a. eventualul constructor cu toţi parametrii luînd valori implicite va fi "rescris" (vom vedea imediat în ce mod) ca un constructor fără nici un parametru;
b. în lipsa constructorului implicit şi a celui cu toţi parametrii luînd valori implicite, se va crea un constructor implicit care să asigure minimul necesar de iniţializări pentru fiecare obiect.

Şi acum, să analizăm două exemple care vor lămuri pe deplin lucrurile.

Exemplul 1

```
// EXEMPLU ERONAT : clasa "punct" nu are constructor fara
// parametri!!!!

// In exemplul urmator vom vedea o rescriere corecta a clasei "punct".

class punct {
    int x,y;
 public :
    punct(int x1=0,int y1=0) {
        x=x1;
        y=y1;
    }
};
```

```
punct polilinie[10]; // aici apare eroarea: desi exista un
                     // constructor cu toti parametri luind
                     // valori implicite compilatorul C++
                     // refuza sa-l foloseasca
void main(void)
{
 // ...
}
```

Exemplul 2

```
// acest exemplu este rescrierea corecta a clasei "punct"
// din exemplul precedent
# include <iostream.h>

class punct {
    int x,y;
    void pct(int x1=0,int y1=0) { x=x1;y=y1; }
 public :
    punct(int x1,int y1) { pct(x1,y1); }

    punct(void) { pct(); }

// metoda de mai jos este necesara pentru initializarea
// sirurilor de obiecte de tip "punct"
    void set_xy(int x1,int y1)
    {
        pct(x1,y1);
    }
};

punct polilinie[10];

void main(void)
{
 for(int i=0;i<5;i++) {
    cout<<"\n\n TASTATI DOI INTREGI :\n ";
    int x,y;
    cin>>x>>y;
    polilinie[i].set_xy(x,y);
 }
}
```

Să analizăm, pe paşi, acţiunile efectuate în cadrul rescrierii:

a. vechiul constructor primeşte numele de "pct" şi este declarat privat;
b. se crează un CONSTRUCTOR IMPLICIT, care-l apelează pe "pct" (fără a-i furniza vreun parametru);
c. se crează un constructor avînd doi parametrii întregi, care vor fi utilizaţi ca parametrii actuali la apelul lui "pct";
d. se crează metoda "set_xy", care va servi la iniţializarea şirurilor de obiecte de tip "punct".

Din cîte se vede, funcţionalitea clasei "punct" nu este afectată, atît că îi creşte - oarecum redundant - numărul de constructori.

În încheiere, vom analiza crearea dinamică a tablourilor de obiecte. Pentru a evidenţia problemele esenţiale, am decis implementarea unei noi clase ("POLILINIE"), care să generalizeze ideea de şir de puncte.

Exemplul 3

```
# include <iostream.h>
# include <process.h>

class punct {
    int x,y;
    void pct(int x1,int y1) { x=x1;y=y1; }
 public :
    punct(int x1,int y1) { pct (x1,y1); }
    punct(void) { pct(0,0); }
};

class PoliLinie {
    int nr_pcte;
    punct * pPoliLinie;
 public:
    PoliLinie(int);
    ~PoliLinie(void);
};

PoliLinie::PoliLinie(int nr)
{
 pPoliLinie=new punct[nr]; //alocarea memoriei necesare
                           // unui sir de "nr" obiecte

// verificam daca alocarea de memorie s-a facut corect
 if(pPoliLinie==NULL)
 {
    cout<<"\n\tMemorie insuficienta !!!";
```

```
        exit(1);
    }
    nr_pcte=nr;
}

PoliLinie::~PoliLinie(void)
{
 delete[nr_pcte] pPoliLinie;// eliberarea memoriei necesa-
                            // re pastrarii unui sir de "nr_pcte"
                            // obiecte
}

void main(void)
{
 PoliLinie poli1(5);
}
```

În primul rînd trebuie remarcat modul de construire a unei polilinii: se alocă dinamic memoria necesară unui şir de "nr" puncte. Acest fapt atrage după sine o consecinţă cu totul aparte, pe care o vom discuta în cele ce urmează. Indiferent că e vorba de o instrucţiune

```
pointerPunct1=new punct;
```

sau

```
pointerPunct2=new punct[100];
```

întotdeauna "new" va returna un pointer la "punct". Din acest motiv, la întîlnirea unor instrucţiuni de genul

```
delete pointerPunct1;
delete pointerPunct2;
```

mecanismul de gestiune a memoriei alocate dinamic nu are de unde să ştie dacă "delete" se referă la eliberarea memoriei aferente unui punct sau a 100 de puncte.

Pentru a rezolva dilema, C++ oferă o soluţie elegantă (vezi destructorul "~ polilinie"). Astfel, prin construcţii de tipul:

```
delete pointerPunct1;
delete[10] pointerPunct2;
```

se va specifica EXPLICIT cantitatea de memorie de eliberat.

3.10.2. "Scurtcircuitarea" mecanismului implicit de gestionare a memoriei alocate dinamic

Spuneam că un compilator nu este capabil să deosebească un pointer la un obiect de un pointer la un şir de 1000 de obiecte. V-aţi întrebat, poate, de ce n-ar fi în stare? Răspunsul este foarte simplu: nefăcînd parte din limbajul propriu-zis, mecanismul de gestiune a memoriei alocate dinamic nu e controlat direct de compilator. Din acest motiv pot să apară erori de genul:

- iniţializarea unor zone de memorie "prin" pointeri pentru care nu s-a alocat încă memorie;
- eliberarea de mai multe ori a aceleiaşi zone de memorie.

La prima vedere, astfel de fenomene ar putea fi interpretate drept slăbiciuni ale limbajului. Dar, să nu uităm, aceste aspecte ţin de faza de execuţie a programului. În plus, nu pot fi ignorate avantajele unei libertăţi cvasi-totale de gestionare a memoriei. E adevarat că această libertate implică multă atenţie din partea programatorului, dar, în acelaşi timp, permite profesioniştilor să exploateze un calculator în maniera cea mai eficientă.

Avantajul modului deschis de a aborda problema memoriei alocate dinamic constă în faptul că permite programatorului să-şi creeze mecanisme proprii în acest domeniu. Astfel, vor putea fi eliminate slăbiciunile mecanismului standard, a cărui prea mare generalitate duce la scăderea vitezei de execuţie a programelor.

Să vedem deci, concret, despre ce este vorba. În limbajul C neobiectual, pentru gestionarea memoriei alocate dinamic, se utilizează funcţii din familia "malloc" şi "free". Modul lor de folosire vă este desigur cunoscut, precum şi faptul că, o dată apelate, programatorul pierde complet controlul a ceea ce se întîmplă în "interiorul" lor. Tot ce ştim este că, din punct de vedere al utilizatorului, comportarea acestor funcţii respectă principiile cutiei negre. Adică, deşi nu-i cunoaştem conţinutul, putem găsi o combinaţie a valorilor bornelor de intrare care să producă la bornele de ieşire valorile dorite. În cazul nostru, cutia neagră va fi funcţia "malloc", dimensiunea în octeţi a zonei de alocat va reprezenta valoarea intrare, iar pointerul la zona alocată va juca rolul de valoare regăsită la bornele de ieşire. Acest mod de abordare a problemei poate fi însă nesatisfăcător în unele cazuri.

Pentru aplicaţiile în timp real, ar fi util un mecanism care să permită un mod mai rapid de alocare a memoriei. Deoarece fiecare apel al unui "malloc" sau "free" este mare consumator de timp, în caz că se alocă frecvent memorie (să zicem de 5000 - 10000 de ori), s-ar putea cîştiga un timp preţios dacă ar exista un mecanism propriu de gestionare a memoriei. Scopul lui ar fi să evite alocarea repetată a memoriei necesare fiecărei entităţi în parte. Acest lucru

s-ar face prin alocarea simultană a memoriei necesare pentru "N" entităţi, cîştigîndu-se astfel timpul consumat la alte "N-1" apeluri. Pînă la epuizarea celor "N" entităţi alocate simultan, mecanismul nostru va acţiona astfel: de cîte ori va fi necesar, va returna cîte un pointer la una din entităţile disponibile. În plus, eliberarea unei entităţi nu va însemna eliberarea memoriei alocate pentru ea, ci adăugarea ei la setul de entităţi disponibile. Astfel, prin scurtcircuitarea mecanismului implicit de gestiune a memoriei, se înţelege înlocuirea lui cu un altul, funcţionînd pe principiile arătate mai sus.

Veţi întreba care ar fi avantajul adus de C++? Graţie constructorilor, destructorilor şi operatorilor "new" şi "delete", totul poate decurge foarte elegant şi eficace. În cadrul acestei subsecţiuni vom discuta doar cîteva principii de bază, care ne vor permite să prezentăm mai tîrziu implementarea unui adevărat mecanism de gestionare a memoriei.

Vom începe cu cîteva observaţii importante legate de variabila "this":

a. În cadrul metodelor ESTE PERMISĂ ATRIBUIREA DE VALORI POINTERULUI"this".
b. La intrarea într-un constructor, "this" va fi diferit de "NULL" DOAR DACĂ S-A ALOCAT DEJA MEMORIE PENTRU ACEL OBIECT!
c. Dacă în cadrul unui constructor SE ATRIBUIE O VALOARE VARIABILEI "this" şi, în plus, la intrarea în constructor,"this" are valoarea "NULL", înseamnă că ACEL CONSTRUCTOR A FOST APELAT DATORITĂ UNEI CREĂRI DINAMICE DE OBIECT (deci cu ajutorul operatorului "new"). Astfel, testînd la intrarea în procedură valoarea lui "this", se va şti cu exactitate tipul apelului.

```
clasa::clasa(void)
{
// pentru ca testul de mai jos sa dea rezultate corecte e OBLIGATORIU
// ca in cadrul constructorului sa existe o atribuire de tipul "this=..."
 if(this==NULL) // APEL DIN "NEW"
    // ...
 else           // OBIECTUL NU A FOST CREAT "DINAMIC"
    // ...
}
```

Înainte de a trece mai departe, să amintim un lucru pe care, pînă acum, l-am trecut cu vederea: alocarea memoriei necesare unui obiect cu ajutorul lui "malloc" NU ADUCE DUPĂ SINE APELUL ÎMPLICIT AL CONSTRUCTORULUI CLASEI RESPECTIVE !!! Acest fapt era de presupus deoarece familia de funcţii "alloc" este păstrată doar pentru compatibilite cu versiunile de C neobiectual. Deci atenţie la secvenţe ca cea de mai jos:

Exemplul 1

```
# include <alloc.h>

class ex{
    int i,j;
 public:
    ex(void){
        i=0;
        j=1;
    }
};

// ATENTIE: se lucreaza cu modelul de memorie HUGE !
void main(void)
{
 ex *p1,*p2;

 p1=(ex *) farmalloc(sizeof(ex));// nu se va apela
                                 // constructorul, ci se
                                 // va aloca doar memorie

 p2=new ex; // se aloca memorie si se apeleaza constructorul
}
```

Revenind la ideea precedentă, să mai facem o observaţie: dacă în cadrul unui constructor se face o atribuire către "this", ORICE MANIPULARE A UNOR MEMBRI AI CLASEI ÎNAINTEA ATRIBUIRII POATE AVEA EFECTE DEZASTRUOASE!

Exemplul 2

```
class clasa {
    int i,j;
 public :
    clasa(void);
};

clasa::clasa(void)
{
 i=16; // POSIBIL DEZASTRU: in caz ca "this==NULL" nu se
       // stie ce zona de memorie va fi initializata!

 if(this==NULL) this=aloca_memorie();
```

```
 i=20; // In acest loc "this" difera sigur de NULL. Zonele de memo-
 j=23; // rie initializate vor fi cele rezervate pentru membrii clasei.
}
```

Din păcate, limbajul C++ nu oferă implicit vreo metodă prin care destructorul unei clase să poată deduce dacă obiectul a fost sau nu creat dinamic. În caz că această informaţie este necesară, clasei i se poate adăuga un membru care să indice acest lucru (vezi exemplul 5). De asemenea, nu există nici o modalitate implicită de a afla dacă un destructor este apelat datorită unui "delete" sau datorită părăsirii domeniului de existenţă a unei variabile.

În schimb, există o metodă graţie căreia poate fi evitat mecanismul standard de eliberare a memoriei: ASIGNAREA VALORII "NULL" POINTERULUI "THIS" !!!! Astfel, dacă într-un destructor apelat de "delete" vom face asignarea "this=NULL", zona de memorie la care pointează "this" nu va mai fi eliberată de către gestionarul standard al memoriei.

Un alt aspect demn de reţinut este legat de modul de alocare şi eliberare a memoriei în cazul unor obiecte create dinamic şi care au ca membrii pointeri la diverse entităţi. Trebuie precizat că, în momentul creării obiectului în cauză, se va aloca memorie DOAR pentru membrii săi, NU şi pentru eventualele entităţi la care ar putea pointa aceştia. Pentru aceste entităţi este necesară o alocare EXPLICITĂ de memorie, eventual chiar în cadrul constructorului. Din acest motiv, eliberarea necesarului de memorie corespunzător unor astfel de entităţi se va face tot EXPLICIT, de obicei în cadrul destructorului clasei. Să analizăm un exemplu practic:

Exemplul 3

```
class CLASA {
     int i;
     CLASA * pCls;
 public :
     CLASA();
     ~CLASA();
};

CLASA::CLASA()
{
// pCls=new CLASA;
//  EROARE: s-ar provoca un apel recursiv infinit al constructorului

// urmeaza varianta corecta de alocare a memoriei pentru
// entitati "pointate" de membri ai unui obiect

 pCls=(CLASA *) new char[sizeof(CLASA)];
```



```
// s-au mai alocat 6 octeti pentru obiectul pointat de "pCls"

 i=0;
 pCls->i=6;
}

CLASA::~CLASA()
{
// delete pCls;
//  EROARE: s-ar provoca apelul recursiv infinit al destructorului

// urmeaza varianta corecta de eliberare a memoriei necesare
// unor entitati "pointate" de membri unui obiect

 delete[sizeof(CLASA)] (char *) pCls;

// s-au eliberat cei 6 octeti la care pointa "pCls"
}

void main()
{
 CLASA * pC;

 pC=new CLASA; // se aloca 6 octeti pentru obiectul nou creat
 delete pC; // se elibereaza cei 6 octeti ocupati de obiect
}
```

Există totuşi o posibilitate de a aloca simultan memorie atît pentru un obiect, cît şi pentru eventualele entităţi la care pointează unii membrii ai săi. Iată cum se poate realiza acest lucru:

Exemplul 4

```
# include <iostream.h>
# include <process.h>

class CLASA {
     int i;
     CLASA * pCls;
 public :
     CLASA();
};

CLASA::CLASA()
{
```

```
if(this!=NULL) {
   cout<<"EROARE : constructorul nu a fost apelat
                   cu NEW";
   exit(1);
}

// se va aloca memoria necesara pastrarii a doua obiecte de tip "CLASA".
// "This" va pointa la primul dintre ele.
 this=(CLASA *) new char[2*sizeof(CLASA)];

 pcls=this+1; // "pCls" va pointa la cel de al doilea obiect de
              // tip "CLASA"

 i=0;                  // se seteaza "this -> i"
 pCls->i=5;            // se seteaza "pCls -> i"
}

void main()
{
 CLASA * pC;

 pC=new CLASA; // la apelul constructorului se vor aloca 12 octeti

 delete pC;// se vor eliberat TOTI cei 12 octeti alocati
}
```

Deosebirea este uşor sesizabilă. În Exemplul 3 se utiliza alocatorul standard de memorie de două ori: mai întîi pentru obiectul propriu-zis, şi apoi pentru cel la care va pointa membrul său "pCls". În Exemplul 4 se va utiliza o singură dată gestionarul memoriei. Dar, de data aceasta, se alocă memorie ÎN ACELAŞI TIMP atît pentru obiect, cît şi pentru entitatea la care pointează "pCls". Datorită acestei stratageme, nu va mai fi necesară prezenţa unui destructor, deoarece se va şti exact cîtă memorie a fost alocată în cadrul constructorului. În momentul întîlnirii lui

```
delete pC;
```

se va elibera întreaga cantitate de memorie.

În încheierea acestei subsecţiuni, vom prezenta un exemplu de utilizare a majorităţii tehnicilor despre care s-a vorbit pînă acum. Vom începe prin a explica pe scurt ce se întîmplă în cadrul programului. S-a implementat o clasă "persoana", ale cărei instanţieri pot fi create în trei moduri diferite. Pentru fiecare mod în parte, constructorul va executa acţiuni specifice, după cum urmează:

a. OBIECT_CREAT_STATIC:

Este vorba de crearea unor obiecte pentru care nu se va aloca dinamic memorie.

b. OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE:

Este vorba de o simulare a tipului de dată "REFERINŢĂ". Pointer-ul "this" al unui obiect "OB" va fi setat la o valoare reprezentînd un pointer la un obiect "OB1" de acelaşi tip, creat deja. În acest mod, orice operaţie efectuată asupra unui membru al lui "OB" va modifica membrul corespunzător al obiectului "OB1", referit de către obiectul "OB". Cu alte cuvinte, "OB" va fi un nou nume al obiectului "OB1".

c. OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE:

Acest mod de creare corespunde situaţiei în care memoria necesară obiectului în cauză nu va fi alocată decît în cadrul constructorului în cauză.

Exemplul 5

```
// Programul de mai jos este compilat cu modelul de memorie "HUGE".
// Pe parcursul lui se va utiliza functia de biblioteca "farcoreleft" pentru
// a se cunoaste cantitatea de memorie disponibila in "heap".

# include <iostream.h>
# include <process.h>
# include <string.h>
# include <alloc.h>

enum mod_de_creare {
 OBIECT_CREAT_STATIC ,
 OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE ,
 OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE
};

class persoana {
     char nume[20];
     char a[1000];
     int  virsta;

     mod_de_creare indicator;
 public:
     persoana(mod_de_creare,char *,int,persoana *);
     ~persoana(void);
};
```

```
// Creaza in unul din cele 3 moduri posibile un obiect de tip persoana
persoana::persoana(mod_de_creare modCreare,
 char *pNume,int _virsta, persoana *pPersoana=NULL)
{
 switch(modCreare) {
 case OBIECT_CREAT_STATIC :
//initializari
    if(this!=NULL) {
       strcpy(nume,pNume);
       virsta=_virsta;
    }
    else {
       cout<<"EROARE: obiect static cu this==
             NULL";
       exit(1);
    }
    break;

 case OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE :
    if(this==NULL)
       if(pPersoana!=NULL)
       {
          this=pPersoana;

          strcpy(nume,pNume); // initializari
          virsta=_virsta;
       }
       else {
// NU EXISTA "PERSOANA" LA CARE SA SE FACA
// "LEGATURA"
          cout<<"EROARE : this ramine NULL\n";
          exit(1);
       }
     else {
// CONSTRUCTORUL NU A FOST APELAT DE UN "NEW"
       cout<<"EROARE : this nu este NULL\n";
       exit(1);
    }
break;

 case OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE :
    if(this==0){
// Urmeaza alocarea memoriei necesare si testarea corectitudinii
```



```
// acestei operatii.
       this=(persoana *)
              new char[sizeof(persoana)];
       if(this==NULL) {
          cout<<"MEMORIE INSUFICIENTA\n";
          exit(1);
       }
//initializari
       strcpy(nume,pNume);
       virsta=_virsta;
       }
       else {
// CONSTRUCTORUL NU A FOST APELAT DIN "NEW"
          cout<<"EROARE : this nu este NULL\n";
          exit(1);
        }
       break;
}

// NOTA: - initializarea ce urmeaza nu se putea face la inceputul metodei
//          deoarece in doua din cazuri "this" ar urma sa-si modifice va-
//          loarea
//        - de remarcat ca in cazul in care obiectul este creat ca
//   OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE,
//          optiunea respectiva va fi inscrisa in cimpul "indicator" al
//          obiectului indicat de "pPersoana" !
 indicator=modCreare;
}

persoana::~persoana(void)
{
// Nu se permite eliberarea memoriei decit pentru obiectele la care nu
// exista "referinte". In cazul obiectelor "referite" de alti pointeri, s e
// scurtcircuiteaza mecanismul standard de eliberare a memoriei (prin
//  "this=NULL").

 if(indicator==
      OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE)
    this=NULL;
}

unsigned long m;
```

```
void mn(void)
{
 persoana p1(OBIECT_CREAT_STATIC,"ion",24);

 persoana *p2 = new persoana(
        OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE,
           "p",34,&p1);

 persoana *p3 = new persoana(
       OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE,
         "v",55);

 persoana *p4 = new persoana(
        OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE,
          "a",43,p3);

 persoana *p5 = new persoana (
        OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE,
         "o",79);

 m=farcoreleft();      // m=60288;

 delete p2;
 m=farcoreleft();      // m= 60288; nu s-a eliberat memoria
                       // alocata deoarece "p2" refera pe "p1"

 delete p3;
 m=farcoreleft();      // m= 60288; nu s-a eliberat memoria
                       // alocata deoarece "p3" e referit de "p4"

 delete p4;
 m=farcoreleft();      // m= 60288; nu s-a eliberat memoria
                       // alocata deoarece "p4" refera pe "p3"

 delete p5;
 m=farcoreleft();      // m= 61328; s-a eliberat memoria
                       // alocata deoarece "p5" nu e referit si nu
                       // refera pe nimeni
}

void main(void)
{
```



```
 m=farcoreleft(); // m=62368;
 mn();
 m=farcoreleft();          // m= 61328;
}
```

Utilizarea unor artificii de genul "OBIECT_CREAT_FARA_ALOCARE_DINAMICA_DE_MEMORIE" trebuie făcută cu foarte mare atenţie, pentru simplul motiv că ele pot scăpa uşor de sub control, avînd spectaculoase efecte distructive. De exemplu, un obiect "O1" referit de "O2" şi "O3" poate fi şters din greşeală, fără a efectua o necesară operaţie de "resetare" a referinţelor la el ("O2" şi "O3"). Dacă, mai tîrziu, vom folosi obiectul "O2", fie se vor citi date eronate, fie se vor modifica locaţii de memorie de la adrese aleatoare.

În cadrul unor programe scurte, scrise de un singur programator, riscurile sînt destul de reduse. La proiecte de dimensiuni mai mari şi la care colaborează mai mulţi programatori, lipsa de coordonare şi consecvenţă poate duce la erori greu de localizat.

Avînd în vedere cele enumerate, vă sugerăm să luaţi exemplul de mai sus drept o simplă gimnastică a minţii. În caz de nevoie, recomandăm utilizarea tipului "referinţă" existent în limbajul C++. Dar aceasta nu înseamnă să nu utilizaţi tehnicile prezentate în cadrul subsecţiunii ori de cîte ori vi se pare necesar.

3.11. Probleme rezolvate.Probleme propuse spre rezolvare

3.11.1. Array-uri de dimensiuni variabile

Mulţi dintre cei care au făcut trecerea de la BASIC la C au avut o vreme nostalgia declarării unor şiruri de dimensiuni variabile, în funcţie de necesităţi. Înainte de a studia modul de abordare a unei astfel de probleme în C++, trebuie să vă facem o recomandare: ar fi bine ca, în urma parcurgerii întregii cărţi, să reveniţi asupra acestui exemplu. Am ţinut să vă atragem atenţia asupra acestui fapt deoarece doar în momentul în care veţi stăpîni şi concepetele de polimorfism şi redefinire a operatorilor vă veţi putea da seama de diferenţa reală între programul de mai jos şi echivalentul său neobiectual. Noul tip de dată se va numi BASIC_ARRAY (în amintirea limbajului după care ne-am inspirat) şi va avea trei metode:

- un constructor (crează şi intializează array-urile);
- un destructor (eliberează memoria alocată pentru un array);

– o funcţie de concatenare a două array-uri (exemplu: dacă avem a={1,2,3} si b={4,5,6}, în urma concatenării lui "a" cu "b" vom obţine a={1,2,3,4,5,6} şi. b={4,5,6}).

Să analizăm acum programul:

```
# include <iostream.h>
# include <process.h>
# include <conio.h>

# define nl "\n"

class BASIC_ARRAY {
 int * pointer;      // pointer la array
 int dimension;      // dimensiune array
 void print(void);   // tiparire array
 void test(void *); // testeaza daca s-a alocat corect memorie
public:
 BASIC_ARRAY(int); // constructor

 ~BASIC_ARRAY(void)    // destructor
  {
    delete[dimension] pointer;
  }

// procedura de concatenare
     void concatenate(BASIC_ARRAY&);
};

// Testeaza valoarea unui pointer pentru care s-a alocat dinamic memorie
// Daca pointer-ul este setat la NULL inseamna ca operatia de alocare
// dinamica a memoriei nu s-a efectuat corect.
void BASIC_ARRAY::test(void * ptr)
{
 if(ptr==NULL)
 {
     cout<<"\n\n\t MEMORIE INSUFICIENTA";
     exit(1);
 }
}

// Creaza un array de intregi avind "nrElem" elemente.
// Initializeaza array-ul cu valori citite de la tastatura.
BASIC_ARRAY::BASIC_ARRAY(int nrElem)
```



```
{
// mesaj explicativ
 cout<<nl<<nl;
 cout<<"Tastati "<<nrElem<<" intregi !"<<nl;

// alocarea memoriei necesare array-ului
 dimension=nrElem;
 pointer=new int[nrElem];

 test((void *) pointer);

// initializare array
 for(int i=0;i<nrElem;i++) cin>>pointer[i];

// tiparire array
  print();
}

// Tipareste un array
void BASIC_ARRAY::print(void)
{
 cout<nl<<nl<<"Sirul este : ";

 for(int i=0;i<dimension;i++)
    cout<<pointer[i]<<" ";
 cout<<nl;

 cout<<"Apasati o tasta !"<<nl;
 getch();
}

// Concateneaza array-ul care a apelat metoda cu cel primit ca
// parametru ( PRIN REFERINTA ! ); rezultatul se va pastra "in"
// obiectul care a apelat metoda

void BASIC_ARRAY::concatenate
                    (BASIC_ARRAY& refARRAY)
{
 int *tmp;
 int final_dimension;

// calcularea noii dimensiuni a rezultatului
 final_dimension=dimension+refARRAY.dimension;
```

```
// alocare memorie necesara
 tmp=new int[final_dimension];

 test((void *) tmp);

// copierea celor doi "operanzi" in cadrul "rezultatului"
for(int i=0;i<final_dimension;i++)
 if(i<dimension) tmp[i]=pointer[i];
 else tmp[i]=refARRAY.pointer[i-dimension];

 delete[dimension] pointer; // eliberarea memoriei folosite
                            // de primul operand

 dimension=final_dimension; // setare dimensiune corecta

 pointer=tmp; // setare "pointer la array"

// tiparire rezultat
 print();
}

void main(void)
{
 BASIC_ARRAY a(3);
 BASIC_ARRAY b(4);

 int i;
 cout<<"Tastati dimensiunea sirului : ";
 cin>>i;
 cout<<nl;

 BASIC_ARRAY c(i);

 a.concatenate(b);
 a.concatenate(c);
}
```

Ca exerciţii propuse spre rezolvare, vă sugerăm să completaţi lista metodelor clasei cu următoarele funcţii:

- "**int find**(BASIC_ARRAY& model)":
  Caută în array-ul iniţial prima apariţie a array-ului "model". Returnează 1 sau 0 după cum modelul există sau nu în cadrul array-ului iniţial.

- "**int replace**(BASIC_ARRAY& model_vechi,
  BASIC_ARRAY& model_nou)":
  Înlocuieşte în cadrul array-ului iniţial toate apariţiile lui "model_vechi" cu "model_nou". Returnează numărul de substituţii efectuate.
- "**int eliminate**(BASIC_ARRAY& model)":
  Elimină din array-ul iniţial toate apariţiile array-ului "model". Returnează 1 sau 0 după cum eliminarea a fost sau nu posibilă.

3.11.2. HASH-TABLE pentru numere de telefon

În această secţiune vom prezenta un concept utilizat deseori de către informaticieni: "hash tables" ("tabele de hashing"). În câteva cuvinte, despre ce este vorba: fie o mulţime de elemente păstrată în memoria calculatorului ca o listă neordonată. A afla dacă o valoare anume este inclusă în această mulţime înseamnă a parcurge SECVENŢIAL lista şi a testa element cu element. Dacă se folosesc diverşi algoritmi de sortare a mulţimii, căutarea se poate face mult mai rapid. Dacă nici astfel viteza nu este suficient de mare, se va îmbina sortarea cu "hashing"-ul.

Ideea de bază în "hashing" este de a încerca să împărţim elementele mulţimii într-un număr oarecare de clase de echivalenţă (având fiecare cam acelaşi număr de elemente - statistic vorbind). Dacă reuşim acest lucru, înseamnă că la căutarea unui element în cadrul mulţimii vom stabili în prima fază cărei clase de echivalenţă îi aparţine, iar apoi îl vom căuta doar în cadrul acelei submulţimi. În cazul în care clasele de echivalenţă vor fi sortate, viteza creşte şi mai mult.

Acestea fiind spuse, să vedem ce-şi propune programul de mai jos. Se doreşte implementarea unei agende telefonice în care căutarea numerelor de telefon să se facă printr-o tabelă de hashing. Criteriul împărţirii în clase de echivalenţă va fi cifra finală a numărului de telefon. În concluzie, vom avea 10 clase de echivalenţă, având aproximativ acelaşi număr de elemente. Tabela de hashing este constituită dintr-un şir de 10 elemente, fiecare conţinând un pointer la o lista simplu înlănţuită. Toate elementele ce aparţin aceleiaşi liste au în comun faptul că se sfîrşesc cu aceeaşi cifră. Deoarece acest exemplu are doar un scop didactic, clasele de echivalenţă nu vor fi sortate. Deci, a căuta în "agendă" un număr de telefon ce se termină cu cifra "i" înseamnă a-l căuta secvenţial în cadrul listei la care pointează al "i"-lea element din tabela de hashing.

Ca exerciţiu, vă propunem spre rezolvare ordonarea claselor de echivalenţă şi căutarea (inserarea) în cadrul acestora prin metoda "divide and conquer" (metoda "înjumătăţirii intervalului"). De asemenea, vă propunem scrierea

unui destructor al clasei "hash_array".

```
# include <stream.h>
# include <process.h>
# include <string.h>
# include <conio.h>

# define ON  0
# define OFF 1

# define nl "\n"

class hash_array {

 struct elem {
     char nume[20];      // nume persoana
     long nr_telefon;    // numar de telefon atasat
     elem *urm;          // pointer la urmatorul element

     elem(long,char*); // constructor element

 }  * sir[10];          // tabela de hashing

 void tipareste(void);      // tiparire agenda telefonica

 void test(void*); // testeaza corectitudinea alocarii
                   // dinamice de memorie
public:
 hash_array(void);  // constructor

// cautare element in agenda
 elem* cauta_numar(long numar_telefon,
                   char *nume="",int insereaza=OFF);

// inserare element in agenda
 elem* insereaza(long numar_telefon,char * nume) {

     // in urma cautarii, elementul se va insera (optiunea ON)
     elem* pElem=cauta_numar(numar_telefon,
                             nume,ON);

     tipareste();
     return pElem;
 }
};

// Initializare element multime
elem::elem(long n,char * p)
{
 nr_telefon=n;
 strcpy(nume,p);
 urm=NULL;
}

// Initializare tabela de hashing (nu exista inca elemente)
hash_array::hash_array(void)
{
 for(int i=0;i<10;i++) sir[i]=NULL;
}

// Testeaza valoarea unui pointer; daca acesta este NULL inseamna ca
// ultima operatie de alocare dinamica de memorie nu s-a efectuat corect
void hash_array::test(void * pointer)
{
 if(pointer==NULL)
 {
     cout<<"\n\n\tMEMORIE INSUFICIENTA !!!";
     exit(1);
 }
}

// Cauta in agenda un numar de telefon . Daca nu-l gaseste si
// parametrul "insereaza" are valoarea "ON", il va insera
elem* _hash_array::cauta_numar(long numar_telefon,
                               char * nume, int insereaza)
{
 elem* tmp,*cap_lista;
 int flag=OFF;          // numarul nu a fost inca gasit

 // determinare "intrare" in tabela de hashing
 int indice=numar_telefon-(numar_telefon/10)*10;
 cap_lista=tmp=sir[indice];

// cita vreme numarul nu a fost gasit si mai exista elemente
// netestate se continua cautarea
 while (tmp!=NULL && flag!=ON)
```

```
    if(tmp->nr_telefon!=numar_telefon)
      tmp=tmp->urm;
    else {
        flag=ON;      // numar gasit
        if(insereaza==ON) {
        cout<<nl<<nl<<nl
         <<"Numarul "<<numar_telefon<<"nu poate fi"
         <<" inserat de doua ori !"<<nl<<nl<<nl;
        return NULL;
    }
}

// daca inscrarea este ceruta si posibila, ca se va face in
// capul listei (ultimul adaugat va fi chiar "intrarea" in tabela)
 if( insereaza==ON ) {
    sir[indice]=new elem(numar_telefon,nume);

    test((void *) sir[indice]);

    sir[indice]->urm=cap_lista;
    return sir[indice];
  }

 return tmp;
}

// Tipareste intreaga agenda telefonica
void hash_array::tipareste(void)
{
 cout<<nl<<nl<<nl<<"AGENDA DE TELEFON : "<<nl<<nl;

 for(int i=0;i<10;i++) {
//  pentru fiecare "intrare" in tabela de hashing ...

    cout<<"Numere ce se sfirsesc cu "<<i
       <<" :"<<nl;
    elem* tmp=sir[i];

    while(tmp!=NULL) { // ... se tipareste clasa de echivalenta
        cout<<tmp->nr_telefon<<" "<<tmp->nume<<nl;
        tmp=tmp->urm;
    }
 }
```



```
 getch();
}

void main(void)
{
 hash_array agenda;

  agenda.insereaza(1110,"Manu");
  agenda.insereaza(1111,"Condas");
  agenda.insereaza(1112,"Gatu");
  agenda.insereaza(1113,"Farna");
  agenda.insereaza(1114,"Bordu");
  agenda.insereaza(1115,"Ionescu");
  agenda.insereaza(1116,"Popescu");
  agenda.insereaza(1117,"Ion");
  agenda.insereaza(1118,"Vasile");
  agenda.insereaza(1119,"Milas");
  agenda.insereaza(6060,"Mocanu");
  agenda.insereaza(2524,"Vintila");
  agenda.insereaza(2610,"Zeda");
  agenda.insereaza(4753,"Istrate");
  agenda.insereaza(7421,"Panait");
  agenda.insereaza(3112,"Ghergu");
  agenda.insereaza(9428,"Pana");
  agenda.insereaza(9694,"Dinu");
  agenda.insereaza(4743,"Burca");
  agenda.insereaza(8231,"Dumitru");

// Urmeaza incercari de inserare a unor elemente deja existente.
// Ele nu vor mai fi efectuate.

  agenda.insereaza(9428,"Pana");
  agenda.insereaza(9694,"Dinu");
  agenda.insereaza(4743,"Burca");
  agenda.insereaza(8231,"Dumitru");

// se efectueaza 10 cautari de numere citite de la tastatura
 for(int i=0;i<10;i++)
 {
    long nr;
    elem * tmp;
    cout<<"Tasteaza un numar de telefon : ";
    cin>>nr;
```

```
    if((tmp=agenda.cauta_numar(nr))!=NULL)
       cout<<"Am gasit : "<<nr<<" "<<
          tmp->nume<<nl<<nl;
    else cout<<"Numarul "<<nr<<" nu exista in
             agenda !"<<nl<<nl;
 }
}
```

3.11.3. Mecanism perfecţionat de gestiune a memoriei

După cum am mai subliniat, unul din aspectele limbajului C++ ce suportă îmbunătăţiri seriose din punct de vedere a vitezei de execuţie, este mecanismul de gestiune al memoriei. În cartea "The C++ Programming Language" ([STRO] p.162-164), Bjarne Stroustrup dă un exemplu de îmbunătăţire a performanţelor mecanismului standard. Atît doar că acel program este prezentat ca o aplicaţie particulară, legată de o clasă anume. Graţie tehnicii de lucru cu "PROJECT"-uri, vom generaliza exemplul lui Stroustrup, aducîndu-i şi sensibile îmbunătăţiri calitative. Programul de mai jos poate fi utilizat cu succes şi în practică, generalitatea lui făcîndu-l util în orice situaţie de folosire intensivă a alocării dinamice de memorie. Ca performanţe atinse de acesta, putem preciza că, faţă de mecanismul implicit, aduce o scădere a timpului consumat cu alocări/eliberări de memorie de circa 19-20%. În cazul unei liste simplu înlănţuite cu 10 elemente cîştigul nu este deloc evident, dar la aplicaţii cu numeroase operaţii de alocare/eliberare cîştigul devine cu adevărat substanţial.

Să vedem care ar fi principiile de bază, după care să analizăm programul propus. Ideea novatoare constă în a aloca SIMULTAN un număr mai mare de entităţi (100 - 1000, în funcţie de aplicaţie), care să fie gestionate apoi de un mecanism propriu. Ce se înţelege prin "gestionate"? În principiu este vorba de:

a. Un alocator: el alocă la un moment dat o listă de N entităţi. De cîte ori programul aplicativ are nevoie de o nouă entitate, se va returna un pointer la următorul element disponibil din listă. În momentul în care lista se goleşte, se va aloca memorie pentru o nouă listă, tot de N elemente.
b. Un eliberator: în loc să elibereze memoria folosită de o entitate, obiectul în cauză se va adăuga ca nou element în capul listei de entităţi disponibile (gestionată de alocator). În acest mod se vor evita secvenţe de operaţii "new" şi "delete".

Un alt avantaj al metodei propuse constă în modul de organizare a clasei "inventar". Un obiect al acestei clase nu este altceva decît un pointer la o listă simplu înlănţuită. Care ar fi inovaţia adusă de noi? În cadrul clasei se va

introduce un membru static ce va contoriza numărul de instanţieri (în cazul nostru numărul de liste simplu înlănţuite). Astfel, nu va fi necesară eliberarea efectivă a tuturor entităţilor alocate DECÎT în momentul în care nu mai există nici o instanţiere a respectivei clase. Şi asta pentru că atîta vreme cît mai există obiecte de acest tip, memoria utilizată de instanţierile distruse nu trebuie eliberată, ci poate fi adaugată în lista de elemente disponibile.

Ca dezavantaj al metodei am putea putea aminti consumul mai ridicat de memorie. Dar, cum dezideratul constă în creşterea vitezei, era de presupus că ea se va face pe seama altui parametru de funcţionare a programului.

Ca temă, propunem o analiză serioasă a genurilor de aplicaţii în care un astfel de gestionar de memorie este eficient (atît ca timp de execuţie, cît şi ca memorie consumată).

În cele ce urmează, vom prezenta pe rînd fişierele programului:

a) **ALOCA_M.H**

```
// contine structurile de date folosite de catre gestionarul de memorie

struct LISTA {
 OBIECT * ob;        // "OBIECT" e definit in "date.h"
 LISTA  * next;
};

class gestionar_memorie {
 LISTA * pLista;
 LISTA * pRecuperate;

public :
 gestionar_memorie();
 ~gestionar_memorie();
 OBIECT * aloca_memorie();
 void elibereaza_memorie(OBIECT *);
};
```

b) **DATE.H**

```
// este folosit ca "include" in "GEST_MEM.CPP"

# include "inventar.h"

// "elem" va fi definit in "inventar.h"
# define OBIECT elem
```

```
# define NUMAR_OBIECTE 4
```

c) GEST_MEM.CPP

```
// contine definitiile metodelor clasei "gestionar_memorie"

# include "date.h"
# include "aloca_m.h"

# include <stdio.h>
# include <process.h>

// Nu exista inca cele doua liste
gestionar_memorie::gestionar_memorie()
{
 pLista=NULL;
 pRecuperate=NULL;
}

gestionar_memorie::~gestionar_memorie()
{
 LISTA * tmp;

 while(pLista!=NULL) {
    delete pLista->ob;
    tmp=pLista;
    pLista=pLista->next;
    delete tmp;
 }

// lista data de "pRecuperate" nu mai trebuie eliberata
// deoarece in acest moment este vida
}

// Daca mai exista obiecte disponibile, va returna un pointer la primul
// dintre ele; in caz contrar face acelasi luc ru,dar alocind in prealabil
// memorie
OBIECT * gestionar_memorie::aloca_memorie()
{
 LISTA * valDeReturnat=pLista;

 if(valDeReturnat==NULL)
// nu exista memorie disponibila; se va aloca memorie pentru un nou "sir"
```



```
{
   LISTA * temporar=(LISTA *)
          new char[NUMAR_OBIECTE*sizeof(LISTA)];
   OBIECT * tmp1=(OBIECT *)
          new char[NUMAR_OBIECTE*sizeof(OBIECT)];

// Se testeaza si "temporar" si "tmp1" deoarece s-ar putea ca
// lungimea lui "OBIECT" sa fie mai mica decit a lui "LISTA";
// in astfel de situatii exista posibilitatea (redusa, ce-i drept)
// ca sa fie suficienta memorie pentru a-crea un "OBIECT", nu
// insa si o "LISTA", si atunci "temporar" va fi NULL, iar "tmp1" nu.
// In toate celelalte situatii ar fi suficient de testat doar "tmp1".

    if( temporar==NULL || tmp1==NULL ) {
       printf("MEMORIE INSUFICIENTA\n");
       exit(1);
    }

// urmeaza ca fiecarui element din sirul "temporar" sa i se
// ataseze memoria necesara cite unui "OBIECT" ( NU UITATI :
// "LISTA" e o lista simplu inlantuita de pointeri la instantieri ale clasei
// "OBIECT" )

    valDeReturnat=temporar++;
    valDeReturnat->next=temporar;
    valDeReturnat->ob=tmp1++;

    for(int i=1;i<NUMAR_OBIECTE-1;i++,tmp1++){
       temporar->ob=tmp1;
       temporar->next=temporar+1;
       temporar=temporar->next;
     } // SFIRSIT "FOR"

    temporar->next=NULL;
    temporar->ob=tmp1;

  } // SFIRSIT"IF"

// primul element al listei initializate se va folosi imediat; de aceea el
// va fi pus chiar acum in "capul" listei cu elemente "de recuperat"

  pLista=valDeReturnat->next;
  valDeReturnat->next=pRecuperate;
```

```
   pRecuperate=valDeReturnat;
   return valDeReturnat->ob;
}

// Va asocia memoria folosita de obiectul disponibilizat unui element din
// lista "de recuperat", pe care-l va elimina din aceasta lista si-l va adauga
// la cea cu obiecte disponibile (in loc sa se elibereze memoria nefolosita,
// ea va fi pusa cu prima ocazie la dispozitia utilizatorului
void gestionar_memorie::elibereaza_memorie(OBIECT*
                                                pObiect)
{
 LISTA * tmp=pRecuperate;

// eliminare din lista "de recuperat"
 pRecuperate=pRecuperate-next;

// reinitializare si adaugare in lista cu elemente disponibile
 tmp->ob=pObiect;
 tmp->next=pLista;
 pLista=tmp;
}
```

d) INVENTAR.H

```
// descrie structurile de date folosite in clasa "inventar"

struct elem {
 char nume_obiect[20];
 int  numar_inventar;
 elem * urmator;
};

class gestionar_memorie;

class inventar {
    elem * cap_lista;

    static gestionar_memorie* gm;
    static int nr_obiecte;

    void distruge_inventar(void);
    void tipareste(void);
```



```
public :
    inventar(void);
    ~inventar(void);
    void adauga_in_lista(char *,int);
    void elimina_din_lista(int);
};
```

e) INVENTAR.CPP

```
// contine definitiile metodelor clasei "inventar"

# include <iostream.h>
# include <process.h>
# include <string.h>
# include <conio.h>

# include "date.h"
# include "aloca_m.h"

# define nl "\n"

# define DA 1
# define NU 0

int inventar::nr_obiecte=0;
gestionar_memorie*
          inventar::gm=new gestionar_memorie;

//Initializari
inventar::inventar(void)
{
 cap_lista=NULL;

 if(gm==NULL) {
    cout<<"MEMORIE INSUFIECIENTA"<<nl;
    exit(1);
 }
 nr_obiecte++;
}

inventar::~inventar(void)
{
 nr_obiecte--;
```

```
 distruge_inventar();
 if(nr_obiecte==0) {
    delete gm;
    cout<<"GATA\n";
 }
}

void inventar::distruge_inventar(void)
{
 while(cap_lista!=NULL)
    elimina_din_lista(cap_lista->numar_inventar);
}

// Atentie: in metoda "adauga_in_lista" nu se face validarea noului
// introdus (nu se verifica unicitatea numarului de inventar) !!!

void inventar::adauga_in_lista(char* nm,int nr)
{
 elem * tmp=gm->aloca_memorie();
 strcpy(tmp->nume_obiect,nm);
 tmp->numar_inventar=nr;
 tmp->urmator=cap_lista;
 cap_lista=tmp;

 tipareste();
}

// Tipareste intreaga lista de elemente
void inventar::tipareste()
{

 elem * tmp=cap_lista;

 cout<<nl<<nl<<"INVENTARUL COMPLET :"<<nl;

 while(tmp!=NULL) {
    cout<<tmp->nume_obiect<<" "<<
        tmp->numar_inventar<<nl;
    tmp=tmp->urmator;
  }
 getch();
}
```



```
// Elimina din lista un obiect identificat prin numarul de inventar
void inventar::elimina_din_lista(int numar)
{
 elem *, tmp=cap_lista,* pDeEliberat;
 int ExistaElement=NU;

 if(cap_lista!=NULL)
    if(cap_lista->numar_inventar==numar) {
       // eliminare cap de lista
       ExistaElement=DA;
       pDeEliberat=cap_lista;
       cap_lista=cap_lista->urmator;
    }
    else
       while(tmp->urmator!=NULL &&
             ExistaElement==NU ) {

          // se cauta secvential elementul dorit
          if(tmp->urmator->numar_inventar==numar){
             ExistaElement=DA;
             pDeEliberat=tmp->urmator;
             tmp->urmator=tmp->urmator->urmator;
          }
          else tmp=tmp->urmator;
       }

 if(ExistaElement==DA) {
    gm->elibereaza_memorie(pDeEliberat);
    tipareste();
 }
}
```

f) M_PRJ.CPP

```
// Programul principal

# include <iostream.h>
# include <process.h>

# include "inventar.h"

void main(void){
 inventar i1,i2, * i3;
```

```
i3=new inventar;

if(i3==NULL)
{
    cout<<"\n\n\tMEMORIE INSUFICIENTA !!!";
    exit(1);
}
cout<<"ADAUGARE :\n";
for(int i=0;i<1000;i++) {
    i1.adauga_in_lista("qwerty",i);
}

cout<<"ELIMINARE :\n";
for(i=0;i<1000;i++) {
    i1.elimina_din_lista(1000-i);
}

cout<<"ADAUGARE :\n";
for(i=0;i<1000;i++) {
    i2.adauga_in_lista("qwerty",i);
}

cout<<"ELIMINARE :\n";
for(i=0;i<1000;i++) {
    i2.elimina_din_lista(1000-i);
}

cout<<"ADAUGARE :\n";
for(i=0;i<1000;i++) {
    i3->adauga_in_lista("qwerty",i);
}

cout<<"ELIMINARE :\n";
for(i=0;i<1000;i++) {
    i3->elimina_din_lista(1000-i);
}

delete i3;
}
```


4. MOŞTENIREA

4.1. Introducere

Vom începe explicarea conceptului de MOŞTENIRE printr-un exemplu legat de familia de funcţii "SCANF" ("scanf", "fscanf" şi "sscanf"). Într-o scurtă paranteză, vom reaminti cîteva caracteristici ale lor:

- **int scanf**(Format,Arg1,Arg2,...,ArgN)

  Citeşte de la dispozitivul standard de intrare "N" valori, în conformitate cu informaţiile din "Format".

- **int fscanf**(Pointer_La_Fisier,Format,Arg1,Arg2,...,ArgN)

  Citeşte din fişierul desemnat de "Pointer_La_Fisier" "N" valori, în conformitate cu informaţiile din "Format".

- **int sscanf**(Pointer_La_Buffer,Format,Arg1,Arg2,...,ArgN)

  Citeşte din bufferul desemnat de "Pointer_La_Buffer" "N"valori, în conformitate cu informaţiile din "Format".

După cum probabil aţi remarcat deja, modul de utilizare a celor trei funcţii este cvasi-identic, variind doar sursa din care vor fi citite valorile în cauză: dispozitivul standard de intrare, un fişier sau un buffer de caractere. O astfel de asemănare frapantă ne poate duce cu gîndul la faptul că, în fond, ar putea exista foarte bine o funcţie generală "_SCANF_", ale cărei proprietăţi sînt "moştenite" de către cele trei funcţii de bibliotecă. Şi, pentru că fiecare dintre procedurile "moştenitoare" are o proprietate caracteristică (sursa de unde preia valorile), sîntem de fapt în faţa a trei tipuri de funcţii aparţinînd uneia şi aceleiaşi FAMILII.

Iată, deci, în cîteva cuvinte la ce se reduce întreg mecanismul de MOŞTENIRE. Avînd o CLASĂ oarecare "B", putem defini o altă CLASĂ "D" care să MOŞTENEASCĂ (sau, dacă preferaţi, să PREIA) TOATE caracteristicile primeia, la care să poată adăuga ALTELE NOI, proprii doar acesteia din urmă. Prima clasă se va numi "CLASA DE BAZĂ", iar cea de a doua "CLASA DERIVATĂ".

Conform specialiştilor firmei BORLAND, "mecanismul de moştenire este, poate, singura mare diferenţă între C şi C++" ([BOR1], p. 141). Cu riscul de a ne repeta, dorim să subliniem încă o dată un lucru deosebit de important: deşi MOŞTENIREA stă la baza altor concepte novatoare (cum ar fi polimorfismul), cheia de boltă a programării obiectuale rămîne ÎNCAPSULAREA. Pentru că, să nu uităm, în lipsa acesteia din urmă, MOŞTENIREA nu ar fi avut nici un suport practic.

4.2. Clase derivate

În cele ce urmează vom da un prim exemplu de derivare a unei clase "hard_disk_2" din clasa "floppy_disk":

Exemplul 1

```
enum stare_operatie { REUSIT , ESEC };
enum stare_protectie_la_scriere { PROTEJAT ,
                                  NEPROTEJAT };

class floppy_disk {

protected:

// cuvintul cheie "protected" permite declararea unor membrii ne-pu-
// blici, care sa poata fi accesati de catre eventualele clase derivate din
// "floppy_disk"

 stare_protectie_la_scriere indicator_protectie;
 int capacitate,nr_sectoare;
public :
 stare_operatie formatare(void);
 stare_operatie citeste_pista(int drive,
          int sector_de_start,
          int numar_sectoare,void * buffer);
 stare_operatie scrie_pista(int drive,
          int sector_de_start,
          int numar_sectoare,void * buffer);
 stare_operatie protejeaza_la_scriere(void);
};

// clasa "hard_disk_1" defineste un hard-disk FARA a profita de
// avantajele mostenirii

class hard_disk_1 {
 stare_protectie_la_scriere indicator_protectie;
 int capacitate,nr_sectoare;
 int  numar_partitii;
public :
 stare_operatie parcheaza_disc(void);
 stare_operatie formatare(void);
 stare_operatie citeste_pista(int drive,
       int sector_de_start,
```



```
   int numar_sectoare,void * buffer);
 stare_operatie scrie_pista(int drive,
   int sector_de_start,
   int numar_sectoare,void * buffer);
 stare_operatie protejeaza_la_scriere(void);
};

// clasa "hard_disk_2" defineste un hard-disk; se utilizeaza
// avantajele mostenirii proprietatilor clasei "floppy_disk"

class hard_disk_2 : public floppy_disk{
 int numar_partitii;
public :
 stare_operatie parcheaza_disc(void);
};

// din definirea metodelor de mai jos reies drepturile de
// accesare ale membrilor clasei derivate

stare_operatie hard_disk_2::parcheaza_disc(void)
{
// CORECT: accesare membru "protected"
 indicator_protectie=PROTEJAT;
// ...
 return REUSIT;
}

void functie(void)
{
 hard_disk_2 hd1;
 hd1.formatare(); // CORECT
 hd1.indicator_protectie=NEPROTEJAT;// EROARE:
                                    // incercare de
                                    // accesare a unui
                                    //  membru protejat
}
```

Incontestabil, cea de a doua metodă de a defini un hard-disk este mult mai elegantă, mai eficace şi mai uşor de înţeles. Să analizăm cu atenţie secvenţa de mai sus. Prin enunţul

```
class hard_disk_2 : public floppy_disk
```

se indică compilatorului următoarele informaţii:

- clasa "hard_disk_2" va fi DERIVATA din clasa "floppy_disk";
- toţi membrii de tip "public" ai clasei "floppy_disk" vor fi moşteniţi (şi deci vor putea fi folosiţi) ca "public" de către clasa "hard_disk_2";
- toţi membrii "protected" ai lui "floppy_disk" vor putea fi utilizaţi ca fiind "protected" în cadrul clasei "hard_disk_2".

Este evident că eficacitatea nu înseamnă doar faptul că în declaraţia clasei derivate nu mai apar informaţiile moştenite (ele fiind automat luate în considerare de către compilator). În plus, nu mai trebuie rescrise nici măcar funcţiile-membru ale clasei de bază, ele putînd fi refolosite EXACT în maniera în care au fost definite. Dar, aşa cum vom vedea în secţiunile viitoare, acest lucru nu înseamnă că, dacă dorim, nu putem să le redefinim, găsindu-le o cu totul altă funcţionalitate.

4.3. Drepturi de accesare a membrilor "moşteniţi"

În secţiunea "3.2." ofeream o sintaxă simplificată a declarării unei clase:

```
specificator_clasa nume_clasa {
 [ [ private : ] lista_membri_1 ]
 [ [ public  : ] lista_membri_2 ]
};
```

De data aceasta vom rafina acea formă, şi vom obţine:

```
specificator_clasa nume_clasa_derivata
[ :     [ modificator_acces_1 ] nume_clasa_baza_1
 [ , [ modificator_acces_2 ] nume_clasa_baza_2 ]
    [ , ... ]
 ]
]

{
 [ [ private   : ] lista_membri_1 ]
 [ [ protected : ] lista_membri_2 ]
 [ [ public    : ] lista_membri_3 ]
} [lista_obiecte];
```

Pentru început, ne vom ocupa de cazul în care există un singur modificator de acces ("modificator_acces_1") şi o singură clasă de bază ("nume_clasa_baza_1"). În secţiunea "4.6." vom trata cazul general (mai mulţi modificatori de acces şi clase de bază), numit şi MOŞTENIRE MULTIPLĂ.

În sintaxa prezentată există trei noutăţi:

- exprimarea relaţiei de "MOŞTENIRE" ("nume_clasa_derivata" fiind DERIVAT din "nume_clasa_baza_1");
- prezenţa lui "modificator_acces";
- apariţia secţiunii "protected".

În funcţie de tipul lui "specificator_clasa" ("class" sau "struct"), "modificator_acces" va lua valoarea IMPLICITĂ "private" sau "public". Rolul acestui "modificator_acces" este ca împreună cu specificatorii "public", "private" sau "protected" întîlniţi în cadrul declarării clasei de bază, SĂ HOTĂRASCĂ dreptul de accesare a membrilor moşteniţi de către o clasă derivată. ATENŢIE: modificatorul de acces NU MODIFICĂ ÎN NICI UN FEL dreptul de accesare a membrilor clasei de bază!

Nu am amintit nimic despre "union"-uri, deoarece ele NU POT FI NICI CLASE DE BAZĂ, NICI CLASE DERIVATE!!!

În continuare, vom prezenta un tabel care sintetizează drepturile de accesare a membrilor unei clase derivate "D" în funcţie de drepturile de accesare a membrilor clasei de bază "B" şi valoarea lui "modificator_acces".

Drept de acces în clasa de bază	**Modificator de acces**	**Drept de acces în clasa derivată**
PUBLIC *PRIVATE* *PROTECTED*	*PUBLIC* *PUBLIC* *PUBLIC*	*PUBLIC* *inaccesibil* *PROTECTED*
PUBLIC *PRIVATE* *PROTECTED*	*PRIVATE* *PRIVATE* *PRIVATE*	*PRIVATE* *inaccesibil* *PRIVATE*

Vom începe explicarea acestor relaţii prin două observaţii legate de cuvîntul cheie "protected":

- el NU poate fi folosit ca "modificator_acces" în cadrul unei relaţii de moştenire;
- rolul său se limitează la a permite accesarea în cadrul unei clase derivate a unor membrii ne-publici ai clasei de bază.

Continuăm comentarea tabelului de mai sus cu o altă remarcă. Indiferent de modificatorul de acces folosit, MEMBRII PRIVAŢI AI UNEI CLASE DE BAZĂ NU VOR PUTEA FI ACCESAŢI DIRECT ÎN CLASA DE-

RIVATĂ! Este o restricţie foarte logică, în lipsa căreia s-ar încălca însăşi ideea de dată privată (adică accesibilă NUMAI metodelor respectivei clase). Dar, NOTA BENE, în ciuda imposibilităţii de a-i accesa direct, ei continuă să EXISTE în cadrul fiecărei instanţieri a clasei derivate. La momentul potrivit, vom vedea că ei pot fi utilizaţi PRIN INTERMEDIUL METODELOR MOŞTENITE de la clasa de bază (vezi exemplul din secţiunea "4.7.").

Şi încă o observaţie: orice "FRIEND" al unei clase derivate are exact aceleaşi drepturi şi posibilităţi de a accesa membrii clasei de bază ca orice alt membru al clasei derivate.

Restul informaţiilor conţinute în tabel nu vor fi comentate, ci exemplificate în cele ce urmează:

Exemplul 1

```
class baza {
 private :
     int a;
     int f1(void);
 protected :
     int b;
     int f2(void);
 public :
     int c;
     int f3(void);
};

class derivata_1 : public baza {
     int a1;
 protected :
     int a2;
 public :
     int a3;
     int fd1(void);
};

class derivata_2 : private baza {
     int a1;
 protected :
     int a2;
 public :
     int a3;
     int fd2(void);
};
```



```
int baza::f3(void)
{
 a=b=c=0; // CORECT: "f3" poate accesa orice membru al clasei
          // "baza"
 return a;
}

// OBSERVATIE:
// In cadrul celor doua metode descrise mai jos ("fd1" si "fd2"),
// DREPTURILE DE ACCESARE a membrilor clasei de baza sint
// IDENTICE. Diferentele apar doar in momentul accesarii
// acestor membri in cadrul unor functii ne-membru.

int derivata_1::fd1(void)
{
 a1=a2=a3=0; // CORECT

 b=f3();    // CORECT: "b" si "f3" sint mostenite cu drept de acces

 c=f2();    // CORECT: "c" si "f2" sint mostenite cu drept de acces

 a=f1();    // EROARE: atit "a" cit si "f1()" sint mostenite,
            // dar nu pot fi accesate in clasa derivata

  return b+c; // CORECT
}

int derivata_2::fd2(void)
{
 a1=a2=a3=0; // CORECT

 b=f3();    // CORECT: "b" si "f3" sint mostenite cu drept de acces

 c=f2();    // CORECT: "c" si "f2" sint mostenite cudrept de acces

 a=f1();      // EROARE: atit "a" cit si "f1()" sint mostenite,
              // dar nu pot fi accesate in clasa derivata

 return b*f2()+c*f3(); // CORECT
}
```

```
void functie(void)
{
 baza        ob_baza;
 derivata_1 ob_der1;
 derivata_2 ob_der2;

 ob_baza.c=ob_baza.a-ob_baza.b;
// EROARE: doar "ob_baza.c" este accesibil in functii ne-membru

 ob_der1.a=ob_der1.b*ob_der1.c/ob_der1.f1()
          +ob_der1.f2()-ob_der1.f3();
 // EROARE: doar "c" si "f3" sint accesibile in functii ne-membru

 ob_der2.b=ob_der2.f3()-ob_der2.a*ob_der2.c/
          ob_der2.f2()*ob_der2.f1();;
// EROARE: nici unul din membrii mosteniti ("a","b","c","f1","f2","f3")
// nu este accesibil in functii ne-membru (in cazul nostru "functie")
}
```

Deoarece o clasă derivată poate fi folosită la rîndul ei ca şi clasă de bază, este necesară o atenţie deosebită în alegerea celei mai bune combinaţii "tip membri"-"modificator acces". În caz contrar, există riscul ca eventuala clasă "nepot" să piardă mult din elasticitate.

O remarcă foarte interesantă ţine de istoricul C++: versiunea iniţială "AT&T 1.00" nu conţinea cuvîntul cheie "protected". Stroustrup argumenta imposibilitatea de a accesa direct dintr-o clasă derivată un membru ne-public al clasei de bază prin încălcarea statutului de membru privat (adică accesibil DOAR clasei ce-l poseda). Totuşi, în cele din urmă, necesitatea de a moşteni şi manipula membri ne-publici ai claselor de bază a dat naştere compromisului numit "protected". Din exemplul de mai jos reies cîteva caracteristici deosebit de importante în utilizarea claselor derivate.

Exemplul 2

```
class BAZA {
 // ...
public :
 int contor1,contor2;

 void f1(void) {
 // ...
 }

 void f2(void) {
 // ...
```



```
 }

 void f3(int a) {
 // ...
 }
};

class DERIVAT1 : public BAZA{
// ...
public :
 int contor1; // definirea unui nou membru, purtind acelasi
              // nume cu cel mostenit

// urmeaza redefinirea metodelor clasei de baza

 void f1(void) {
 // ...
    BAZA::f1();      // apelul metodei "f1" a clasei "BAZA"
 }

 int f2(void) {
 // ...
    BAZA::f2();      // apelul metodei "f2" a clasei "BAZA"
    return 0;
 }

 void f3(void)
 {
    BAZA::f3(10);    // apelul metodei "f3" a clasei "BAZA"
 }
};

class DERIVAT2 : BAZA{ // implicit, modificatorulde acces
                       // folosit este "private"
// ...
public :
 BAZA::contor1;
 BAZA::f1;     // cei doi membri mosteniti sint FORTATi sa
               // REDEVINA "public"
};

void main(void)
{
```

```
DERIVAT1 d1;
DERIVAT2 d2;

d1.f1();
d1.f2();            // apelul metodelor redefinite

d1.BAZA::f1();      // apelul metodei clasei "BAZA"
d1.contor1=0;       // initializarea noului membru
d1.BAZA::contor1=1;     // initializarea membrului mostenit

d2.f1();
d2.contor1=0;       // utilizarea membrilor mosteniti si
                    // FORTATI sa REDEVINA "public"

d2.f2();            // EROARE: membru privat inaccesibil
d2.contor2=1;       // EROARE: membru privat inaccesibil
}
```

În primul rînd vom analiza clasa "DERIVAT1". Se observă că, în cadrul acestei clase, vor fi redefinite ("OVERLOADED") toate cele trei metode ale clasei de bază. O facilitate de limbaj permite însă accesarea de către orice membru al clasei "DERIVAT1" a tuturor acestor metode aparţinînd clasei "BAZA" şi redefinite în "DERIVAT1". Modul de folosire este foarte simplu: funcţia în cauză va fi prefixată cu numele "clasei posesoare" (în cazul nostru "BAZA") urmat de "::". În lipsa prefixării cu "BAZA::", efectul va fi ori un apel recursiv infinit (în cazul metodelor "f1" si "f2"), ori o eroare la compilare (este cazul lui"f3", care e apelată cu un parametru întreg, deşi a fost redefinită ca neavînd nici un parametru).

Trebuie amintit încă un lucru: deşi în clasa "DERIVAT1" am adăugat încă o variabilă membru "contor1", în funcţia "main" pot fi accesate AMBELE variabile "contor1" (atît cea moştenită, cît şi cea adaugată). Diferenţierea lor se va face prin prefixarea variabilei moştenite cu "BAZA::".

Vom mai sublinia că prefixarea cu numele clasei de bază este valabilă şi în afara corpului metodelor (vezi cazurile instrucţiunilor "d1.BAZA::f1();" şi "d1.BAZA::contor1=1;").

A doua noutate este legată de clasa "DERIVAT2". Exprimarea, oarecum ciudată

```
public :
 BAZA::contor1;
 BAZA::f1;
```


nu ascunde nimic altceva decît posibilitatea de a forţa unii membrii să redevină de tip "public". Acest lucru va fi făcut ÎN CIUDA REGULILOR DE MODIFICARE A DREPTULUI DE ACCES din cadrul mecanismului de moştenire (deci nu este nicidecum vorba de adăugări de noi membrii). Aşa cum se vede, atît în cazul variabilelor cît şi în cel al metodelor, este suficientă simpla precizare a numelui şi a clasei de proveniență. De altfel, încercarea de a da mai multe amănunte despre respectivul membru (tipul variabilei, respectiv valoarea returnată sau tipul parametrilor) va fi semnalizată ca eroare de compilare. De remarcat faptul că, în acelaşi mod, se poate efectua şi forţarea membrilor ce şi-au pierdut calitatea de a fi "protected" (datorită unei moşteniri de tip privat). Există şi o restricţie: indiferent de modificatorul de acces utilizat, NU ESTE PERMISĂ transformarea unui membru PRIVAT al clasei de bază într-unul PUBLIC sau PROTECTED al celei derivate.

O altă chestiune ce trebuie discutată constă în posibilitatea de a se efectua conversii de tipul "B*"->"D*" şi "D*"->"B*", unde prin "D" am notat o clasă derivată din clasa "B". Exemplul de mai jos ne va lămuri pe deplin asupra problemei.

Exemplul 3

```
class baza{
 // ...
};

class derivat_1 : public baza{
 // ...
};

class derivat_2 : private baza{
 // ...
};

void main(void)
{
 baza      *  pb,ob;
 derivat_1 *  pd1,od1;
 derivat_2 *  pd2,od2;

 pb=&od1;     // CORECT: conversie implicita

 pb=&od2;     // EROARE: nu se poate face conversie implicita

 pd1=&ob;     // EROARE: nu se poate face conversie implicita
 pd2=&ob;     // EROARE: nu se poate face conversie implicita
```

```
    pb=(baza*) &od2; // CORECT: conversie explicita

    pd1=(derivat_1*) &ob; // CORECT: conversie explicita
    pd2=(derivat_2*) &ob; // CORECT: conversie explicita
}
```

Reluînd, în cuvinte, cele prezentate în cadrul exemplului, conversia "DERIVAT*"->"BAZA*" se poate face:

IMPLICIT: – dacă "DERIVAT" moşteneşte pe "BAZA" în manieră publică;

EXPLICIT: – dacă "DERIVAT" se obţine din "BAZA" prin moştenire în manieră privată.

Conversia "BAZA*"->"DERIVAT*" nu poate fi făcută DECÎT EXPLICIT, prin operatorul "cast".

Să analizăm următoarea situaţie: o variabilă publică a clasei "baza" devine privată în urma moştenirii ei de către clasa "derivat_2". Făcînd o conversie "derivat_2*"->"baza*", printr-un pointer la "baza" vom putea accesa respectiva variabilă ca şi cum ar fi publică. Cu toate că acest lucru este POSIBIL, nu este DELOC RECOMANDABIL. Înainte de a trece mai departe, vom relua avertismentul lui Bjarne Stroustrup: ÎNCĂLCAREA DREPTURILOR DE ACCES PRIN CONVERSII EXPLICITE între pointeri la tipuri de date legate între ele prin relaţia de moştenire riscă să ducă la erori aproape imposibil de detectat. De aceea, în astfel de situaţii, se recomandă o atenţie deosebită.

4.4. Constructori şi destructori

Avînd în vedere că scopul claselor derivate este acela de a permite adăugarea unor noi caracteristici unei clase de bază, este evident că, în general, clasele derivate vor avea nevoie de constructori mai complecşi decît cele de bază. Pentru a uşura cît mai mult munca programatorului şi pentru a avea o comportare cît mai naturală, constructorii claselor de bază vor fi apelaţi IMPLICIT, dacă acest lucru este posibil. În caz contrar, va fi necesară specificarea EXPLICITĂ a parametrilor în cauză (vezi exemplul de mai jos). Specificarea explicită a parametrilor constructorului se poate face şi dacă programatorul doreşte apelul unui constructor anume din setul de constructori ai respectivei clase. Deoarece o clasă derivată nu este altceva decît o construcţie "în jurul" unei clase de bază, este logic ca mai întîi să se apeleze constructorii claselor de bază, şi doar după aceea să se execute acţiunile specifice constructorului clasei derivate.

În cazul destructorilor, problema se va pune exact invers: mai întîi va fi apelat destructorul clasei derivate şi abia după aceea cel al clasei de bază.

Exemplul 1

```
# include <iostream.h>
# include <conio.h>

class BAZA
{
 protected :
    int cont;
 public :

    void SET_CONTOR(int val) { cont=val; }
    void TYPE_CONTOR(void) {
       cout<<"\n\n CONTOR = "<<cont;
    }

    BAZA(int a=0)
    {
       SET_CONTOR(a);
       cout << "\nConstructorul clasei BAZA;\t
             CONTOR="<<cont<<"\n";
    }

    ~BAZA(void)
    {
       cout << "\nDestructorul clasei BAZA;\t
           CONTOR="<<cont<<"\n";
    }
};

class DERIVAT : public BAZA
{
public :
 DERIVAT(void)     // se va apela IMPLICIT "BAZA(0)" !
 {
    cout << "Constructorul clasei DERIVAT\n";
 }

 DERIVAT(int b) : BAZA(b) // apel explicit al constructorului
                          // clasei de baza
 {
```

```
        cout << "Alt constructor al clasei DERIVAT\n";
    }

    void TYPE_CONTOR(void)
    {
        cout<<"\n\n Contorul clasei derivate este "
            <<cont;
    }

    ~DERIVAT(void)
    {
        cout << "\nDestructorul clasei DERIVAT;";
    }

};

void main(void)
{
 clrscr();

 BAZA    b1,b2=1;
 DERIVAT d1,d2(2),d3(3);

 BAZA    * pb1;
 DERIVAT * pd1;

 pb1=&b2;
 pd1=&d2;

 pb1->TYPE_CONTOR();
 pd1->TYPE_CONTOR();

 cout<<"\n";
}
```

Putem urmări ordinea executării procedurilor programului ghidîndu-ne după succesiunea mesajelor afişate pe ecran:

Constructorul clasei BAZA; CONTOR=0

Constructorul clasei BAZA; CONTOR=1

Constructorul clasei BAZA; CONTOR=0
Constructorul clasei DERIVAT

Constructorul clasei BAZA; CONTOR=2
Alt constructor al clasei DERIVAT

Constructorul clasei BAZA; CONTOR=3
Alt constructor al clasei DERIVAT

CONTOR = 1

Contorul clasei derivate este 2

Destructorul clasei DERIVAT;
Destructorul clasei BAZA; CONTOR=3

Destructorul clasei DERIVAT;
Destructorul clasei BAZA; CONTOR=2

Destructorul clasei DERIVAT;
Destructorul clasei BAZA; CONTOR=0

Destructorul clasei BAZA; CONTOR=1

Destructorul clasei BAZA; CONTOR=0

La al doilea constructor al clasei derivate se vede foarte bine modul în care se vor specifica parametrii necesari constructorului clasei de bază. De notat doar că exprimarea

```
DERIVAT(int b) : BAZA(b)
```

nu era necesară, ea putînd fi înlocuită de mai simpla

```
DERIVAT(int b) : (b)
```

În acest moment trebuie să atragem atenţia asupra unui aspect prezentat în exemplul de mai jos.

Exemplul 2

```
class B
{
  B();      // EROARE: constructor inaccesibil clasei D
 ~B();      // EROARE: destructor inaccesibil clasei D
};

class D : B
{
};
```

Analizînd cele două structuri de date, tragem următoarea concluzie: ESTE INTERZIS CA ÎN CADRUL UNEI CLASE BAZA SĂ AVEM TOŢI CONSTRUCTORII SAU DESTRUCTORII PRIVAŢI. Avînd în vedere gravitatea unor astfel de situaţii (în care clasa derivată NU POATE ACCESA CONSTRUCTORUL SAU DESTRUCTORUL clasei de bază), erorile de acest gen vor fi semnalizate în faza de compilare.

Deşi pare fără sens, existenţa unor constructori şi destructori privaţi nu este lipsită de interes. Nu uitaţi, aceste metode vor putea fi folosite - de exemplu - în cadrul unor funcţii "friend".

Cu titlu de observaţie, subliniem că destructorii şi constructorii claselor de bază NU SE MOŞTENESC. Cu toate acestea, accesibilitatea lor este esenţială în procesul creării şi distrugerii instanţierilor clasei derivate.

Încheiem această secţiune prin analiza creării dinamice de obiecte fără a utiliza mecanismul implicit de gestionare a memoriei. Pentru înţelegerea exemplului ce urmează trebuie să mai facem o subliniere. Dacă în cadrul constructorului unei clase derivate se face o atribuire de genul "this=...", constructorul clasei de bază nu va fi apelat pînă în momentul în care atribuirea a fost făcută. Lucrurile se petrec astfel deoarece valoarea lui "THIS" în cadrul constructorului clasei de bază va fi una şi aceeaşi cu cea atribuită lui în constructorul clasei derivate. Reciproc, dacă în cadrul constructorului clasei de bază i se atribuie lui "THIS" o valoare, ea va fi folosită în continuare şi în constructorul clasei derivate!!!

Exemplul 3

```
# include <iostream.h>
# include <process.h>

class BAZA {
public :

 BAZA(void * pointer=NULL)
 {
    cout<<"\n\t\t La apelul constructorului BAZA()
         this="<<this;

    if(this==NULL)
       if(pointer!=NULL)
          this=(BAZA *) pointer;
       else
       {
          cout<<"\nEROARE : in \"BAZA()\" se
              forteaza this=NULL";
          exit(1);
       }
    else this=this; // Nu este inutil: vezi observatia de
                    // la sfirsitul exemplului !

    cout<<"\n\t\t La iesirea din BAZA() this="
        <<this;
 }
};

class DERIVAT : public BAZA{
public :

 DERIVAT(void * pointer=NULL)
 {
    cout<<"\n\t\t La apelul constructorului
        DERIVAT() this="<<this;

    if(this==NULL)
       if(pointer!=NULL)
          this=(DERIVAT *) pointer;
       else
       {
          cout<<"\nEROARE: in \"DERIVAT()\" se
              forteaza this=NULL";
          exit(1);
       }
    else this=this; // Nu este inutil: vezi observatia de
                    // la sfirsitul exemplului !

    cout<<"\n\t\t La iesirea din DERIVAT() this="
        <<this<<"\n\n";
 }
};

void main(void)
{
 cout<<"OBIECTE CREATE FARA ALOCARE DINAMICA DE
      MEMORIE :\n\n";
 cout<"\tBAZA";
 BAZA b1;
 cout<<"\n\tDERIVAT\n";
 DERIVAT d1;
```

```
cout<<"OBIECTE CREATE PRIN ALOCARE DINAMICA DE
     MEMORIE :\n\n";
cout<<"\tBAZA";
new BAZA(&b1);

cout<<"\n\tDERIVAT\n";
new DERIVAT(&d1);

// Fiecare din cele patru creari dinamice de obiecte ce apar in continuare
// ar provoca oprirea executiei programului. Acest fapt s-ar datora
// atingerii apelurilor functiei "exit" din cadrul celor doi constructori
// ("BAZA" si "DERIVAT").

// new BAZA(NULL);
// new BAZA();
// new DERIVAT(NULL);
// new DERIVAT();

}
```

În urma execuţiei acestui program, pe ecran se vor tipări următoarele mesaje:

OBIECTE CREATE FARA ALOCARE DINAMICA DE MEMORIE :

BAZA

La apelul constructorului BAZA() this=8FD3:0FFE
La iesirea din BAZA() this=8FD3:0FFE

DERIVAT

La apelul constructorului DERIVAT() this=8FD3:0FFC
La apelul constructorului BAZA() this=8FD3:0FFC
La iesirea din BAZA() this=8FD3:0FFC
La iesirea din DERIVAT() this=8FD3:0FFC

OBIECTE CREATE PRIN ALOCARE DINAMICA DE MEMORIE :

BAZA

La apelul constructorului BAZA() this=0000:0000
La iesirea din BAZA() this=8FD3:0FFE

DERIVAT

La apelul constructorului DERIVAT() this=0000:0000
La apelul constructorului BAZA() this=8FD3:0FFC
La iesirea din BAZA() this=8FD3:0FFC
La iesirea din DERIVAT() this=8FD3:0FFC

Ceea ce merită într-adevăr analizat în acest exemplu este constructorul clasei DERIVAT. Se observă că "IF"-ul care testează valoarea lui "this" are - oarecum surprinzător - două ramuri. Pe una se face atribuirea necesară în caz că "THIS" este NULL, iar pe cealaltă se va face o atribuire de tipul

```
this=this;
```

Deşi nu are nici un scop aparent, această atribuire este FUNDAMENTALĂ în cazul în care constructorul poate fi apelat atît cu "THIS!=NULL", cît şi cu "THIS==NULL". Vom justifica afirmaţia prin faptul că, dacă în cadrul constructorului clasei derivate se va detecta (LA COMPILARE) o atribuire către "THIS", apelul constructorului clasei de bază se va face DOAR DUPĂ EFECTUAREA ATRIBUIRII. Ei bine, de aici apare şi problema. S-ar putea ca atribuirea unei valori către "this" să se facă DOAR PE O SINGURĂ "RAMURĂ" A PROGRAMULUI (de exemplu, într-un IF se face doar pe "THEN", nu şi pe "ELSE"). În cazul în care execuţia va avea loc pe o altă ramură, apelul constructorului clasei de bază nu se va mai executa DEOARECE NU S-A ATINS INSTRUCŢIUNEA CONŢINÎND ATRIBUIREA CĂTRE "THIS". Iată deci o problemă cu totul neaşteptată, şi care poate da multă bătaie de cap programatorului neatent!

4.5. Metode virtuale. Utilizarea listelor eterogene

Prin definiţie, un tablou omogen este un tablou conţinînd elemente de acelaşi tip (de exemplu, un şir de întregi). După cum am spus, un pointer la o clasă "B" poate păstra adresa oricărei instanţieri a vreunei clase derivate din "B". Deci, avînd un şir de pointeri la obiecte de tip "B", înseamnă că, de fapt, putem lucra şi cu tablouri neomogene. Astfel de tablouri neomogene se vor numi ETEROGENE. Una din caracteristicile limbajului C++ constă tocmai în faptul că MECANISMUL FUNCŢIILOR VIRTUALE PERMITE TRATAREA UNIFORMĂ (sau, dacă preferaţi, ÎN ACEEAŞI MANIERĂ) a tuturor elementelor unui masiv de date ETEROGENE! Acest lucru este posibil datorită unor facilităţi ce ţin de asocieri făcute DOAR ÎN MOMENTUL EXECUŢIEI PROGRAMULUI, nu în momentul compilării!

Vom face o scurtă paranteză pentru a explica mai pe larg aceste principii generale. Să presupem că avem o clasă "B" care posedă o metodă publică "M". Din clasa "B" vom deriva mai multe clase "D1", "D2", "D3",..., "Dn". Dacă clasele

```
    {
        pB=p; t=tp;
    }

    void tipareste_valoare(void)
    {
        if( t== _BAZA_ )
          pB->tipareste_valoare();
        else
          if( t== _DERIVAT_1 )
             ((DERIVAT_1 *) pB)->
                      tipareste_valoare();
          else
             ((DERIVAT_2 *) pB)->
                      tipareste_valoare();
    }
};

void main(void)
{
 BAZA     a[2];
 DERIVAT_1 b1;
 DERIVAT_2 b2;

 LISTA_ETEROGENA  p[4];

 a[0].set_val(1);
 a[1].set_val(2);
 b1.set_val(3);
 b2.set_val(4);

 p[0].set(&a[0]);

 p[1].set(&b1,_DERIVAT_1);// se face o conversie IMPLI-
                          // CITA "DERIVAT_1"-> "BAZA*"

 p[2].set((BAZA *) &b2,_DERIVAT_2); // se face o conver-
                 // sie EXPLICITA "DERIVAT_2"-> "BAZA*"

 p[3].set(&a[1]);

 for(int i=0;i<4;i++)
    p[i].tipareste_valoare();
```



```
}
```

În urma execuţiei programului, pe ecran se vor afişa următoarele mesaje:

Element BAZA cu VALOARE = 1
Element DERIVAT_1 cu VALOARE = 3
Element DERIVAT_2 cu VALOARE = 4
Element BAZA cu VALOARE = 2

Pentru a reuşi o tratare uniformă a celor trei tipuri de obiecte a fost necesară crearea unei a patra clase, numită LISTA_ETEROGENA. Aceasta va păstra atît pointer-ul la obiectul respectiv, cît şi o informaţie suplimentară, referitoare la tipul obiectului referit de pointer. Tipărirea informaţiilor semnificative ale fiecărui obiect se va face în metoda "LISTA_ETEROGENA::tipareste_valoarea". În această funcţie membru sînt necesare o serie de teste pentru apelul metodei corespunzînd fiecărui tip de obiect. Vă daţi seama că, dacă în loc de 3 tipuri de obiecte am lucra cu 10 sau 100, soluţia ar deveni mult mai greoaie şi ne-elegantă.

Reamintim că în instrucţiunea

```
p[2].set((BAZA *) &b2,_DERIVAT_2);
```

operatorul "CAST" este absolut necesar. Conform celor afirmate în secţiunea "4.3.", dacă dintr-o clasă "B" derivăm în maniera privată o altă clasă "D", conversia "D*"->"B*" nu se poate realiza decît de maniera EXPLICITĂ.

În Exemplul 2 vom vedea o modalitate mult mai simplă şi elegantă de a trata OMOGEN un masiv de date eterogene. În Exemplul 1 tratarea omogenă se referea la instrucţiunea "for", în care, indiferent de tipul elementului curent, se va apela una şi aceeaşi funcţie-acţiune. Se vede foarte bine că, în acest caz, este vorba de o pseudo tratare uniformă, deoarece decizia de tratare diferenţiată este aminată de la nivelul lui "for" la cel al metodei "LISTA_ETEROGENA::tipareste_valoarea". Iată acum o rezolvare tipic obiectuală a problemei.

Exemplul 2

```
# include <iostream.h>

class BAZA {
 protected :
    int valoare;
 public :
    void set_val(int a) { valoare=a; }

    virtual void tipareste_valoare(void)
    {
```

```
        cout<<"Element BAZA cu VALOARE = "
            <<valoare<<"\n";
      }
};

class DERIVAT_1 : public BAZA {
 public :
    void tipareste_valoare(void)
    {
        cout<<"Element DERIVAT_1 cu VALOARE = "
            <<valoare<<"\n";
    }
};

class DERIVAT_2 : BAZA {
 public :
    BAZA::set_val;

    void tipareste_valoare(void)
    {
        cout<<"Element DERIVAT_2 cu VALOARE = "
            <<valoare<<"\n";
    }
};

class LISTA_ETEROGENA {
    BAZA * pB;
 public:
    void set(BAZA * p) { pB=p; }

    void tipareste_valoare(void)
    {
       pB->tipareste_valoare();
    }
};

void main(void)
{
 BAZA    a[2];
 DERIVAT_1 b1;
 DERIVAT_2 b2;

 LISTA_ETEROGENA p[4];
```



```
 a[0].set_val(1);
 a[1].set_val(2);
 b1.set_val(3);
 b2.set_val(4);

 p[0].set(&a[0]);
 p[1].set(&b1);
 p[2].set((BAZA *) &b2);
 p[3].set(&a[1]);

 for(int i=0;i<4;i++)
    p[i].tipareste_valoare();
}
```

Şi de astă dată, pe ecran vor fi afişate aceleaşi informaţii ca şi în cazul exemplului precedent:

Element BAZA cu VALOARE = 1
Element DERIVAT_1 cu VALOARE = 3
Element DERIVAT_2 cu VALOARE = 4
Element BAZA cu VALOARE = 2

Marea diferenţă între cele două exemple constă în aceea că, acum, clasa LISTA_ETEROGENA are o metodă "tipareste-valoarea" mult simplificată. În noua funcţie membru nu mai este necesară nici testarea tipului de pointer memorat şi nici conversia pointerului memorat la tipul original. Şi toate acestea doar din cauza prefixării metodei "BAZA::tipareste_valoarea" cu cuvîntul cheie "virtual". Practic, în urma întîlnirii lui "virtual", compilatorul va lua AUTOMAT şi FĂRĂ NICI UN ALT AJUTOR DIN PARTEA PROGRAMATORULUI toate deciziile necesare pentru efectuarea unui "LATE BINDING" corect.

Să încercăm să înţelegem modul de realizare al unui mecanism atît de puternic. Prefixarea unei metode "F" a clasei "B" cu cuvîntul cheie "virtual" are următorul efect: toate clasele ce o moştenesc pe "B" şi nu REDEFINESC funcţia "F", o vor moşteni întocmai. Să presupunem acum că o clasă "D", derivată din "B", va REDEFINI metoda "F". În această situaţie, COMPILATORUL VA AVEA SARCINA DE A ASOCIA UN SET DE INFORMAŢII SUPLIMENTARE, CU AJUTORUL CĂRORA SE VA PUTEA DECIDE (IN MOMENTUL EXECUŢIEI) CARE METODĂ "F" VA FI APELATĂ.

Observaţii

a. Deoarece funcţiile virtuale necesită memorarea unor informaţii suplimentare, instanţierile claselor care au metode virtuale vor ocupa în memorie mai mult loc decît ar fi necesar în cazul în care nu ar exista decît metode

ne-virtuale.

Trebuie amintite şi două restricţii: nici FUNCŢIILE NE-MEMBRU şi nici METODELE STATICE NU POT FI DECLARATE VIRTUALE!

b. Pentru ca mecanismul metodelor virtuale să poată funcţiona, metoda în cauză NU POATE FI REDECLARATĂ în cadrul claselor derivate ca având aceiaşi parametrii şi returnînd un ALT TIP de dată. În schimb, este permisă REDECLARAREA cu alt set de argumente, dar cu ACELAŞI tip de dată returnat. ATENŢIE: în acest caz noua funcţie NU VA MAI PUTEA FI MOŞTENITĂ MAI DEPARTE CA FIIND METODĂ VIRTUALĂ!

c. Evitarea mecanismului de apel uzual pentru funcţiile virtuale se face foarte uşor prin prefixarea numelui funcţiei cu numele clasei aparţinătoare urmat de "::".

Exemplul 3

```
void DERIVAT::f1(void)
{
 BAZA::tipareste_valoare(); // apelul metodei din clasa
                            //  BAZA
 tipareste_valoare(); // apelul metodei din clasa DERIVAT
}
```

d. Cu toate că nici un constructor nu poate fi declarat virtual, destructorii acceptă o astfel de prefixare. Ea poate fi deosebit de utilă în cazul în care se doreşte distrugerea uniformă a unor masive de date eterogene. O situaţie clară de utilizare a unor destructori virtuali vom obţine transformînd Exemplul 2 în următoarea manieră:

Exemplul 4

```
# include <iostream.h>

class BAZA {
 protected :
    int valoare;
 public :
    virtual void set_val(int a) { valoare=a; }

    virtual void tipareste_valoare(void)
    {
       cout<<"Element BAZA cu VALOARE = "
           <<valoare<<"\n";
    }
    virtual ~BAZA(void)
    {
       cout<<"\t\t DESTRUCTOR BAZA\n\n";
    }
};

class DERIVAT : BAZA {
 public :
    BAZA::set_val;

    void tipareste_valoare(void)
    {
       cout<<"Element DERIVAT cu VALOARE = "
           <<valoare<<"\n";
    }

    ~DERIVAT(void)
    {
       cout<<"\t\t DESTRUCTOR DERIVAT\n";
    }
};

class LISTA_ETEROGENA {
    BAZA * pB;
 public:

    BAZA* get(void)
    {
       return pB;
    }

    void set(BAZA * p)
    {
       pB=p;
    }

    void tipareste_valoare(void)
    {
       pB->tipareste_valoare();
    }
};
```

```
void main(void)
{
 LISTA_ETEROGENA  p[4];

 p[0].set(new BAZA);
 p[1].set((BAZA *) new DERIVAT);
 p[2].set((BAZA *) new DERIVAT);
 p[3].set(new BAZA);

 for(int i=0;i<4;i++)
 {
    (p[i].get())->set_val(i);
    p[i].tipareste_valoare();
    delete p[i].get();
 }
}
```

În urma execuţiei acestui program se vor afişa pe ecran următoarele date:

Element BAZA cu VALOARE = 0
DESTRUCTOR BAZA

Element DERIVAT cu VALOARE = 1
DESTRUCTOR DERIVAT
DESTRUCTOR BAZA

Element DERIVAT cu VALOARE = 2
DESTRUCTOR DERIVAT
DESTRUCTOR BAZA

Element BAZA cu VALOARE = 3
DESTRUCTOR BAZA

În acest mod avem confirmarea clară a corectitudinii distrugerii uniforme a listei eterogene de obiecte.

c. O funcţie virtuală "F", redefinită într-o clasă derivată "D", va fi vazută în noua sa formă de către toate clasele derivate din "D". Această redefinire nu afectează cu nimic faptul că ea este virtuală, metoda rămînînd în continuare virtuală şi pentru eventualele clase derivate din "D".

Exemplul 5

```
# include <iostream.h>

class B1 {
 public:
    virtual void t1(void) { cout<<"B1\n"; };
```



```
};

class B2 : public B1 {
 public :
    void t1(void) { cout<<"B2\n"; }
};

class B3 : public B2 {
};

void main(void)
{
 B1 a;
 B2 b;
 B3 c;

 a.t1();
 b.t1();
 c.t1();
}
```

Efectul execuţiei acestui exemplul va fi tipărirea pe ecran a următoarelor şiruri de caractere:

B1
B2
B2

Aşadar, în clasa "B3", funcţia virtuală "t1" este "vazută" aşa cum a fost ea redefinită în clasa "B2". Vechea formă (cea corespunzătoare clasei "B1") poate fi utilizată aşa cum am arătat la punctul "c." al observaţiilor.

f. Să considerăm exemplul de mai jos:

Exemplul 6

```
# include am.h

class BAZA {
 protected :
    virtual void tipareste_mesaj(void)
    {
       cout<<"\tMetoda apartine clasei BAZA\n";
    }

 public :
    void functie(void)
    {
```

```
        cout<<"\n\n URMEAZA UN MESAJ :\n";
            tipareste_mesaj();
    }
};

class DERIVAT : public BAZA {

 void tipareste_mesaj(void)
 {
    cout<<"\tMetoda apartine clasei DERIVAT\n";
 }
};

void main(void)
{
 BAZA    a;
 DERIVAT b;

 BAZA *  pB;

 a.functie();
 b.functie();

 pB=&a;
 pB->functie();

 pB=&b;
 pB->functie();
}
```

În urma execuţiei sale se va afişa următorul text:

URMEAZA UN MESAJ :
Metoda apartine clasei BAZA

URMEAZA UN MESAJ :
Metoda apartine clasei DERIVAT

URMEAZA UN MESAJ :
Metoda apartine clasei BAZA

URMEAZA UN MESAJ :
Metoda apartine clasei DERIVAT

Ce se observă în acest exemplu? Metoda"BAZA::functie" va fi moştenită

de către clasa DERIVAT. Mecanismul de "LATE BINDING" permite selectarea corectă a lui "DERIVAT::tipareste_valoare" la fiecare apel al lui "functie" pentru cîte o instanţiere a clasei DERIVAT.

g. Cînd se pune problema de a decide dacă o metodă să fie sau nu virtuală, programatorul va lua în considerare următoarele aspecte:
   - În cazul în care contează doar performanţele unui program şi se exclude intenţia de a-l mai dezvolta în continuare, SE VA OPTA PENTRU METODE NE-VIRTUALE.
   - În cazul programelor ce urmează să suporte dezvoltări ulterioare, situaţia este cu totul alta. Vor fi declarate virtuale toate acele metode ale clasei "CLASA" care se consideră că ar putea fi redefinite în cadrul unor clase derivate din "CLASA" şi, în plus, modificarea va trebui să fie sesizabilă la "nivelul" lui "CLASA"! Toate celelalte metode pot rămîne ne-virtuale.

4.6. Moştenire multiplă

Una din marile noutăţi ale versiunii AT&T 2.0 a limbajului C++ constă în posibilitatea ca o clasă să fie derivată din mai multe clase de bază. În versiunile precedente, relaţia de moştenire era limitată la doua clase: cea de BAZĂ şi cea DERIVATĂ. Cu toate acestea, moştenirea multiplă se putea simula destul de uşor:

Exemplul 1

```
class B1
{
 // ...
};

class B2
{
 // ...
};

class B3
{
 // ...
};

class D1 : B1
{
 B2 b1;
```

```
 B3 b2;
};
```

Din punct de vedere al structurii de date, clasa"D1" din exemplul de mai sus este echivalentă cu forma:

```
class D1 : B1 , B2 , B3
{
 // ...
};
```

Din punct de vedere funcţional, există totuşi destule diferenţe (modul de apel al metodelor claselor membre, dificultăţi în lucrul cu masive de date eterogene etc.). Tocmai de aceea, noua versiune elimină aceste neajunsuri, acceptînd existenţa relaţiei de moştenire multiplă. În acest mod, o clasă derivată va putea avea SIMULTAN mai multe clase de bază!

Există, totuşi, şi unele restricţii:

a. O clasă derivată "D" nu poate moşteni direct de două ori aceeaşi clasă "B". Motivul este foarte simplu: nu s-ar putea face distincţia între membrii purtînd acelaşi nume!

Exemplul 2

```
class B
{
 // ...
};

class D : B , B    // EROARE !!!!
{
};
```

b. O clasă derivată "D" nu poate moşteni simultan o clasă "B" şi o alta "D1", derivată (direct sau nu) tot din "B". De asemenea, nu pot fi moştenite simultan două clase "D1" şi "D2" aflate în relaţia de moştenire "B"->... ->"D1"->...->"D2".

Exemplul 3

```
class B
{
 // ...
};

class C1 : B
{
};
```



```
class C2 : C1
{
};

class D1 : B , C2    // EROARE
{
};

class D2 : C1 , C2   // EROARE
{
};
```

În schimb, o moştenire de genul

```
class B
{
 // ...
};

class D1 : B
{
};

class D2 : B
{
};


class D : D1 , D2
{
};
```

este găsită corectă de către compilatorul C++.

Pentru a realiza o moştenire multiplă, este suficient să specificăm după numele clasei derivate o listă de clase de bază separate prin virgulă (evident, aceste nume pot fi prefixate cu modificatorii de acces "public" sau "private"). Dar atenţie: ordinea claselor de bază nu este indiferentă! Apelul constructorilor respectivi se va face EXACT în ordinea enumerării numelor claselor de bază! Reluînd observaţia de la moştenirea simplă, apelul constructorilor tuturor claselor de bază se va efectua ÎNAINTEA eventualelor iniţializări de variabile-membru. Mai mult, aşa cum vom vedea în exemplul de mai jos, această ordine nu poate fi modificată nici chiar de ordinea apelului explicit al constructorilor claselor de bază în cadrul constructorului clasei derivate.

Exemplul 4

```
class B1
{
    char c;

 public :
    B1(char c1) : c(c1) {};
};

class B2
{
    int c;

 public :
    B2(int c1) : c(c1) {};
};

class D : B1 , B2
{
    char a,b;

 public :
    D(char a1,char b1) : a(a1) , b(b1)
      B2((int) a1) , B1(b1)
 {
    // ...
 }
};

void main(void)
{
 D d(3,3);
}
```

După cum am mai afirmat, indiferent de ordinea iniţializării variabilelor şi de cea a constructorilor specificaţi explicit, în definirea constructorului clasei derivate

```
D(char a1,char b1) : a(a1) , b(b1)
      B2((int) a1) , B1(b1) {};
```

succesiunea operaţiilor va fi următoarea: apelul constructorului "B1", apoi "B2" şi, doar la sfîrşit, iniţializarea lui "a" şi "b".

O situaţie cu totul particulară o au aşa-zisele CLASE VIRTUALE.

Pentru a avea o imagine cît mai clară asupra problemei, să analizăm exemplul de mai jos:

Exemplul 5

```
class B
{
};

class D1 : B
{
};

class D2 : B
{
};

class M1 : D1 , public D2
{
};

class M2 : virtual D1 , virtual public D2
{
};
```

În urma moştenirii claselor "D1" şi " D2", clasa "M1" va îngloba două obiecte de tipul "B" (cîte unul pentru fiecare din cele două clase de bază). În practică, ar putea exista situaţii în care este de dorit moştenirea unui singur obiect de tip "B". Pentru aceasta se va proceda ca şi în cazul clasei "M2": se vor prefixa cele două clase de bază cu cuvîntul cheie "virtual". În urma acestei decizii, clasa "M2" nu va conţine decît O SINGURĂ INSTANŢIERE A CLASEI "B".

Deşi în aparenţă totul e deosebit de simplu, se mai poate pune o întrebare: cărei clase îi va aparţine acea unică instanţiere? Lui D1 sau lui D2? INSTANŢIEREA ÎN CAUZĂ VA APARŢINE LUI "D1", adică PRIMEI CLASE VIRTUALE (din lista claselor de bază) care a fost derivată din "B". În urma analizei exemplului de mai jos veţi remarca un fapt destul de ciudat, pe care-l vom comenta în continuare.

Exemplul 6

```
class B
{
 public:
     int i;
};
```

```
class D1 : public B
{
};

class D2 : public B
{
};

class M1 : public D1 , public D2
{
};

class M2 : virtual public D1 , virtual public D2
{
};

class M3 : virtual public D2 , virtual public D1
{
};

void main(void)
{
 M1 m1;
 M2 m2;
 M3 m3;

 m1.D1::i=0;
 m1.D2::i=1;

 m1.i=0; // EROARE: exprimare ambigua ( de care din cei
         // doi membri "i" e vorba ? )

 m2.D1::i=4;      // CORECT

 m2.D2::i=5;      // ATENTIE: desi nu se semnalizeaza eroare la
                  // compilare, aceasta instructiune NU
                  // ARE NICI UN EFECT !!!

 m2.i=6;          // CORECT: e echivalent cu "m2.D1::i= 6;"

 m3.D1::i=7;      // ATENTIE: desi nu se semnalizeaza eroare la
                  // compilare, aceasta instructiune NU
                  // ARE NICI UN EFECT !!!
```



```
 m3.D2::i=8;      // CORECT

 m3.i=9;          // CORECT: e echivalent cu "m3.D2::i= 9;"
}
```

Concluzia este următoarea: deşi conform regulii enunţate mai sus variabila membru aparţine instanţierii primei clase din lista claselor de bază, cele două forme "m2.D2::i" şi "m3.D1::i" nu vor fi tratate de către compilator ca şi erori. În schimb, efectul lor nu va fi cel scontat, adică atribuirile în cauză nu vor fi efectuate (instrucţiunile respective neavînd deci nici un efect). Atenţie deci la această sursă de erori! Cea mai simplă modalitate de a evita astfel de neplăceri constă în a renunţa la prefixarea variabilei membru cu clasa de care aparţine. În acest mod nu pot apărea ambiguităţi, deoarece variabila este unică.

Definirea claselor virtuale ne obligă să modificăm regulile de apelare a constructorilor claselor de bază. În cele ce urmează vom prezenta noul set de reguli:

R1. *Constructorii claselor de bază virtuale vor fi apelaţi ÎNAINTEA celor corespunzînd unor clase de bază ne-virtuale.*

R2. *În caz că o clasă posedă mai multe clase de bază virtuale, constructorii lor vor fi apelaţi în ordinea în care clasele au fost declarate în lista claselor de bază. Apoi vor fi apelaţi (tot în ordinea declarării) constructorii claselor de bază ne-virtuale, şi abia în cele din urmă constructorul clasei derivate.*

R3. *Chiar dacă în ierarhia claselor de bază există mai mult de o instanţiere a unei clase de bază virtuale, constructorul respectiv va fi apelat doar o singură dată.*

R4. *Dacă, în schimb, în aceeaşi ierarhie există atît instanţieri virtuale cît şi ne-virtuale ale unei aceleiaşi clase de bază, constructorul va fi apelat:*

O SINGURĂ DATĂ	– pentru toate instanţierile virtuale;
DE "N" ORI	– pentru cele ne-virtuale (dacă există "N" instanţieri de acest tip).

Pentru a clarifica lucrurile vom oferi un exemplu mai complex, pe care vom studia modul de aplicare a acestor reguli.

Exemplul 7

```
int i=0;

class B1
{
 public:
    B1(void) {
       i++;
```

```
    }
};

class B2
{
 public:
    B2(void) {
        i++;
    }
};

class D1 : public B1 , virtual public B2
{
 public:
    D1(void) {
        i++;
    }
};

class D2 : public B1 , virtual public B2
{
 public:
    D2(void) {
        i++;
    }
};

class M : public D1 , virtual public D2
{
 public:
    M(void) {
        i++;
    }
};

void main(void)
{
 M m1;
}
```

Ordinea în care vor fi apelaţi constructorii este următoarea:

```
B2();
B1();
```



```
D2();
B1();
D1();
M();
```

Ea corespunde întru totul regulilor enunţate mai sus. În încercarea de a construi o instanţiere a clasei "M", se vor analiza cele două clase de bază. Deoarece "D2" este clasa de bază virtuală, se va începe prin a crea un obiect de tip "D2" (regula "R1"). Pentru aceasta va fi necesară crearea unor obiecte de tip "B1" şi "B2". Se va începe cu "B2" deoarece este vorba tot de o clasă de bază virtuală (aceeaşi regulă "R1"). Din aceste motive, primul constructor apelat va fi "B2", urmat apoi de "B1". În acest moment va putea fi apelat şi constructorul "D2", deoarece au fost deja create instanţierile claselor sale de bază (regula "R2"). În fine, pentru a crea un obiect de tipul "D1" ar trebui să se înceapă cu crearea unui obiect de tip "B2". Cum clasa de bază "B2" este virtuală, şi, în plus, există deja o instanţiere a acestei clase, se va renunţa la crearea obiectului de tip "B2" (regula "R3"). Se va continua, deci, cu apelul constructorului clasei de bază "B1", urmat apoi de constructorul "D1" (regula "R2"). Deoarece în acest moment sînt create instanţierile claselor de bază "D1" şi "D2", se va putea apela şi constructorul clasei "M" (tot regula "R2").

În încheierea secţiunii prezentăm încă un exemplu de funcţionare a mecanismului de moştenire multiplă.

Exemplul 8

```
# include <graphics.h>
# include <conio.h>

class arc_de_cerc
{
    int xc,yc,raza,unghi_start,unghi_sfirsit;
 public:
    arc_de_cerc(int x,int y,int r,int u1, int u2)
    {
        xc=x;
        yc=y;
        raza=r;
        unghi_start=u1;
        unghi_sfirsit=u2;

        arc(xc,yc,unghi_start,unghi_sfirsit,raza);
    }
}
```

```
class linie
{
    int x1,x2,y1,y2;
 public:
    linie(int a,int b,int c,int d)
    {
        x1=a;
        y1=b;
        x2=c;
        y2=d;

        line(x1,y1,x2,y2);
    }
};

class cap : public arc_de_cerc
{
 public :
    cap(int xc,int yc,int u_st,int u_sf,int raza)
     : arc_de_cerc(xc,yc,u_st,u_sf,raza)
    {
    }
};

class ochi_stinga  : public arc_de_cerc
{
 public :
    ochi_stinga(int xc,int yc,int u_st,int u_sf,
     int raza) : arc_de_cerc(xc,yc,u_st,u_sf,raza)
    {
    }
};

class ochi_dreapta : public arc_de_cerc
{
 public :
    ochi_dreapta(int xc,int yc,int u_st,int u_sf,
     int raza) : arc_de_cerc(xc,yc,u_st,u_sf,raza)
    {
    }
};
```



```
class gura : public arc_de_cerc
{
 public :
    gura(int xc,int yc,int u_st,int u_sf,
     int raza) : arc_de_cerc(xc,yc,u_st,u_sf,raza)
    {
    }
};

class fata_umana : cap,ochi_stinga,
                   ochi_dreapta,gura,linie
{
 public:
    fata_umana(void) : cap(100,100,50,0,360),
                linie(100,100,100,110),
                ochi_stinga(80,90,3,0,360),
                ochi_dreapta(120,90,3,0,360),
                gura(100,100,40,240,300)
    {}
};

void main(void)
{
 int gd,gm;

 detectgraph(&gd,&gm);
 initgraph(&gd,&gm,"\\compiler\\t_pascal\\bgi");

 fata_umana f;

 getch();
 closegraph();
}
```

Neputînd forţa clasa "fata_umana" să moştenească de patru ori clasa "arc_de_cerc", vom recurge la un subterfugiu: din ea vom deriva alte patru clase ("cap", "ochi_stinga", "ochi_dreapta" şi "gura"), pe care le vom utiliza ca şi clase de bază ale clasei "fata_umana".

4.7. Problemă rezolvată. Exerciţii propuse spre rezolvare

În cele ce urmează, vom prezenta un exemplu care are drept scop fa-

miliarizarea cu situaţii de o complexitate mai apropiată de cea reală. Ideea programului este de a implementa o familie de ferestre grafice, care să faciliteze şi să facă mai plăcut dialogul cu utilizatorul.

În esenţă, programul constă într-un PROJECT compus din 4 module: clasele "settings", "general_window", "error_window" şi fişierul "main.cpp" (conţine programul principal care utilizează cele trei clase). Vom începe să analizăm pe rînd fişierele în cauză, primul fiind "header"-ul clasei "settings":

/*FISIERUL STTNG_CL.DAT */

```
class settings {

 int cul,bkcul;       // culoare de scriere si de fond
 struct textsettingstype txt; // atributele de scriere ale
                              // textelor
 struct fillsettingstype fll; // atributele de umplere

protected:
     void put_settings(void); // setare atribute mod grafic
     void get_settings(void); // salvare atribute mod grafic

};
```

Acest "header" va fi inclus în fişierul "SETTINGS.CPP", care conţine definiţiile metodelor declarate în "STTNG_CL.DAT". Clasa "settings" are ca scop implementarea unor rutine de salvare/restaurare a caracteristicilor modului de lucru grafic (culoare de desenare şi fond, caracteristici de scriere şi de umplere).

/*FISIERUL SETTINGS.CPP */

```
# include <graphics.h>
# include "sttng_cl.dat"

// Restaureaza atributele modului de lucru salvate anterior
void settings::put_settings(void)
{
 settextstyle(txt.font,txt.direction,txt.charsize);
 settextjustify(txt.horiz,txt.vert);

 setfillstyle(fll.pattern,fll.color);
 setcolor(cul);
 setbkcolor(bkcul);
}
```



```
// Salveaza atributele curente de lucru in mod grafic
void settings::get_settings(void)
{
 gettextsettings(&txt);
 getfillsettings(&fll);
 cul=getcolor();
 bkcul=getbkcolor();
}
```

În continuare, vom analiza descrierea unei clase numită "GENERAL_WINDOW", din care să putem deriva mai apoi toate tipurile de ferestre dorite. Vom începe cu "header"-ul care conţine declaraţia clasei:

/*FISIERUL G_WIN_CL.DAT */

```
# include "sttng_cl.dat"

# define MARGINE 8

class general_window : settings {

private:
//.....VARIABILE MEMBRU.....

 int x,y,dx,dy; // coordonatele colturilor stinga-sus si
                // dreapta-jos ale ferestrei

 void * store_image; // pointer la imaginea peste care
                     // va fi desenata fereastra

 int image_size;    // lungimea (octeti) zonei de mai sus

 char * window_name; // pastreaza numele ferestrei

 int name_size; // lungimea numelui ferestrei (nr. caractere)

//.....METODE.....

 void set_xy(void); // initializeaza coordonatele colturilor
                    // ferestrei
 void get_image(void); // salvare imagine de sub fereastra
 void draw_window(void); // desenare fereastra
```

```
    int bad_coord(int,int,int,int); // verifica coord. colturi

  protected :

    void my_settings(int,int); // seteaza caracteristici mod
                               // de lucru grafic

    settings::get_settings;    // forteaza la "protected" cele
    settings::put_settings;    // doua metode mostenite

  public :

    void create_window(char *,int=-1,int=-1,
                       int=-1,int=-1);
    void destroy_window(void);
  };
```

Fişerul "GNRL_WIN.CPP" va conţine definiţiile metodelor clasei "general_window".

/*GNRL_WIN.CPP*/

```
  # include <iostream.h>
  # include <graphics.h>
  # include <process.h>
  # include <string.h>

  # include "g_win_cl.dat" // contine declaratia clasei

  // Salvare imagine de "sub" fereastra
  void general_window::get_image()
  {
    image_size=imagesize(x,y,dx,dy);
    store_image=new char[image_size];
    if (store_image==NULL) {
       cout<<"Out of memory in GENERAL_WINDOW\n";
       exit(1);
    }
    getimage(x,y,dx,dy,store_image);
  }

  // Setarea caracteristicilor modului de lucru grafic
  void general_window::my_settings(int bk_color,
                                   int color )
```



```
{
// APEL METODA MOSTENITA "get_settings()" !!!
 get_settings(); // salvarea caracteristici mod grafic

 settextstyle(DEFAULT_FONT,HORIZ_DIR,1);
 settextjustify(LEFT_TEXT,LEFT_TEXT);

 setfillstyle(SOLID_FILL,bk_color);
 setcolor(color);
}

// Desenare fereastra
void general_window::draw_window()
{
// afisarea numelui ferestrei
 bar(x,y,dx,y+9);
 outtextxy((dx-x)/2+x-textwidth(window_name)/2,
           y+9,window_name);

// desenarea corpului ferestrei
 setcolor(WHITE);
 setfillstyle(SOLID_FILL,BLACK);
 bar(x,y+9,dx,dy);
 rectangle(x,y+10,dx,dy);
 rectangle(x+2,y+12,dx-2,dy-2);
}

// Seteaza abscisele si ordonatele colturilor stinga-sus
// (x,y) si dreapta-jos (dx,dy) la valorile implicite
void general_window::set_xy(void)
{
 x=getmaxx()/2-150;
 y=getmaxy()/2-80;
 dx=getmaxx()/2+150;
 dy=getmaxy()/2+80;
}

// Verifica daca ambele colturi sint in cadrul ecranului si
// daca intr-adevar coltul stinga-sus este mai sus si mai in
// stinga decit cel dreapta-jos
int general_window::bad_coord(int x1,int y1,
                              int x2,int y2)
{
```

```
 if( x1<0 || y1<0 || x2<0 || y2<0 || x2>getmaxx()
    || y2>getmaxy() || x1>=x2 || y1>=y2 )
 return 1;

 return 0;
}

// Va crea o fereastra cu numele dat de "window_name" si cu
// coordonatele colturilor stinga-sus si dreapta-jos luind
// valorile parametrilor "i1"-"i4" (eventual se pot utiliza
// valorileimplicite);
void general_window::create_window(char *win_name,
                                   int x1, int y1,
                                   int x2,int y2)
{
 if( bad_coord(x1,y1,x2,y2) )
// setare coord. colturi la val. implicite
    set_xy();
 else {
// init. coordonatelor la valorile transmise ca parametri
    x=x1;
    y=y1;
    dx=x2;
    dy=y2;
 }

 get_image();      // salvare imagine de "sub" fereastra

 my_settings(RED,BLACK);   // seteaza caracteristicile de
                           // lucru in mod grafic

 int lung=textwidth(win_name);

// daca numele fereastrei nu incape in fereastra, fereastrei
// nu i se va afisa numele
 if (dx-x-2*MARGINE>=lung) {
// numele incape in fereastra
    name_size=strlen(win_name)+1;
    window_name=new char[name_size];
    strcpy(window_name,win_name);
 }
 else {
```



```
// numele nu incape in fereastra
    name_size=2;
    window_name=new char[name_size];
    strcpy(window_name," ");
 }

 draw_window();   // desenare fereastra

 put_settings(); // restaurare optiuni utilizator
}

// - restaurare caracteristici mod de lucru grafic folosite de utilizator;
//  - refacere imagine de "sub" fereastra;
// - eliberare memorie consumata pentru pastrarea imaginii salvate.
void general_window::destroy_window(void)
{
 put_settings();
 putimage(x,y,store_image,COPY_PUT);
 delete[image_size] store_image;
 delete[name_size] window_name;
}
```

În fine, clasa "ERROR_WINDOW" va avea următorul "header":

/* E_WIN_CL.DAT */

```
# include "g_win_cl.dat"

class error_window : general_window {
 public :
    void create_error_window(char *);
    void destroy_error_window(void);
};
```

Definirea metodelor respective se va face în fişierul "ERR_WIN.CPP":

/* ERR_WIN.CPP */

```
# include <graphics.h>
# include <process.h>
# include <conio.h>

# include "e_win_cl.dat" // contine declaratia clasei

# define MARGINE 8
```

```
void error_window::create_error_window(char *
                        error_msg)
{
 char * mesaj,msg[]="Mesaj de eroare prea lung !";
 int lung,inalt,x=getmaxx()/2,y=getmaxy()/2;

// setare caracteristici de lucru
 my_settings(BLACK,WHITE);

 lung=textwidth(error_msg)/2;
 inalt=textheight(error_msg);
 mesaj=error_msg;

 // daca mesajul este prea lung va fi inlocuit cu unul de avertizare
 if(lung+MARGINE>x) {
     lung=textwidth(msg);
     inalt=textheight(msg);
     mesaj=msg;
 }

// inainte de a crea fereastra se refac vechile caracteristici
// (pentru a nu fi pierdute prin supra-inscriere)
 put_settings();

 create_window("EROARE",x-lung-MARGINE,y-2*inalt,
               x+lung+MARGINE,y+3*inalt);

 my_settings(BLACK,WHITE);
 outtextxy(x-lung,y+2*inalt,mesaj);
 put_settings();
}

void error_window::destroy_error_window()
{
 destroy_window();
}
```

Referitor la metoda "error_window::destroy_error_window()", este clar că ea putea lipsi, în locul ei utilizindu-se "destroy_window()"-ul moştenit. S-a preferat această soluţie deoarece ni se pare mai elegant ca numele metodei să fie cît mai sugestiv.

În fine, fişierul "MAIN.CPP" arată astfel:

/* MAIN.CPP */

```
# include <conio.h>
# include <stdio.h>
# include <process.h>
# include <graphics.h>

# include "e_win_cl.dat"

// Asigura initializarea modului de lucru grafic si furnizarea
// mesajelor de avertizare si eroare corespunzatoare
void init_graph(void)
{
 int gd,gm,err;

 detectgraph(&gd,&gm);
 initgraph(&gd,&gm,"\\borlandc\\bgi");
 err=graphresult();
 if(err!=grOk) {
    printf("GRAPHICS ERROR : %s\n",
          grapherrormsg(err));
    exit(1);
 }
}

// Pe ecran se va crea un desen oarecare (pentru a se arata ca
// utilizarea ferestrelor nu distruge imaginea existenta)
void creaza_scena()
{
 char txt[] ="Este";
 char txt1[]="un program";
 char txt2[]="demonstrativ al cartii de C++";

 circle(50,50,50);
 circle(80,80,80);
 circle(120,120,120);

 settextjustify(LEFT_TEXT,LEFT_TEXT);

 settextstyle(TRIPLEX_FONT,HORIZ_DIR,10);
 outtextxy((getmaxx()/2)-(textwidth(txt)/2),
           (getmaxy()/3)+(textheight(txt)/2),txt);
```

```
 settextstyle(TRIPLEX_FONT,HORIZ_DIR,2);
 outtextxy((getmaxx()/2)-(textwidth(txt1)/2),
           getmaxy()-2*textheight(txt1),txt1);

 settextstyle(GOTHIC_FONT,HORIZ_DIR,4);
 outtextxy((getmaxx()/2)-(textwidth(txt2)/2),
           getmaxy()-2,txt2);
}

// Aceasta procedura va crea si manipula diverse tipuri de ferestre,
// incercind sa surprinda situatii cit mai complexe (suprapuneri partiale
// sau totale, etc.)
void actiune(void)
{
 general_window gw1;
 general_window gw2;
 general_window gw3;

 gw1.create_window("Fereastra numarul 1");
 outtextxy(500,30,"A");
 gw2.create_window("Numarul 2",10,10,100,100);
 outtextxy(500,60,"B");
 gw3.create_window("Numarul 3",50,50,150,150);
 outtextxy(500,90,"C");

 getch();

 gw3.destroy_window();
 getch();
 gw2.destroy_window();
 getch();
 gw1.destroy_window();
 getch();

 outtextxy(500,120,"D");

 gw1.create_window("Fereastra numarul 1");
 gw2.create_window("Numarul 2",10,10,100,100);
 gw3.create_window("Numarul 3",50,50,150,150);

 error_window ew1;
 ew1.create_error_window("Disc protejat la
                          scriere !");
```



```
 outtextxy(500,150,"E");
 getch();

 error_window ew2;
 ew2.create_error_window("Memorie insuficienta !");
 outtextxy(500,180,"F");
 getch();

 ew2.destroy_error_window();
 getch();
 ew1.destroy_error_window();
 getch();
}

void main(void)
{
 init_graph();

 creaza_scena();
 actiune();

 closegraph();
}
```

Ca probleme propuse spre rezolvare, am sugera extinderea acestui set de ferestre, după cum urmează:

a. crearea unui mecanism de redimensionare şi schimbare a poziţiei ferestrei generale pe ecran;
b. crearea unei ferestre de prompt (citire de caractere, numere întregi şi reale);
c. crearea unei ferestre de căutare pe disc a unor clase de fişiere (Ex: *.*, *.txt, a?n.*, ...).

5. ALTE FACILITĂŢI ALE LIMBAJULUI C+ +

5.1. Funcţii pure şi clase abstracte

Începem cu această temă deoarece este direct legată de unul dintre subiectele precedentului capitol, şi anume de mecanismul de moştenire a metodelor virtuale. Vom reaminti în cîteva cuvinte despre ce era vorba. Dacă o metodă "M" a unei clase de baza "B" este prefixată cu cuvîntul cheie "virtual", ca va fi utilizată identic de toate clasele "MB" ce o moştenesc pe "B" fără a redefini metoda "M". Dacă o clasă "MB1", derivată din "B", va redefini metoda "M", compilatorul va şti să utilizeze noua metodă indiferent dacă ea va fi manipulată:

a. de un obiect de tipul MB1;
b. prin intermediul unui pointer la obiecte de tipul B, care memorează adresa unui obiect de tip MB1.

Oricum, este clar că din moment ce pot fi apelate, metodele virtuale vor fi atît declarate, cît şi definite.

În unele situaţii practice se poate găsi o clasă "B" din care se derivează toate celelalte clase ale ierarhiei. Uneori clasa "B" este atît de generală, încît unele metode ce reprezintă caracteristici fundamentale ale ei nu pot fi descrise la acest nivel, ci doar în clasele derivate.

De exemplu, fie următoarea ierarhie de clase:

VIEŢUITOARE:

- LALELE
- NUFERI
- CAI
- OAMENI

Să presupunem că în cadrul clasei VIEŢUITOARE există o metodă numită NUTRIŢIE. La acest nivel, funcţia NUTRIŢIE nu poate fi definită decît abstract. Deşi putem spune că NUTRIŢIA este "totalitatea proceselor de ingerare şi asimilare a hranei" ([MDLR], p. 488), această definiţie nu ne dă nici o informaţie despre modalităţile practice de realizare a procesului respectiv. NUTRIŢIA va fi însă foarte uşor de definit concret în cadrul claselor derivate. Spre exemplu, procesele de tipul masticaţiei sau digestiei

omeneşti pot fi descrise pînă în cele mai mici amănunte. Cînd spunem "O VIEŢUITOARE SE HRĂNEŞTE", ne vom închipui un nufăr, un cal sau un om. Deci, cînd spunem "O VIEŢUITOARE", ne referim la o entitate abstractă (deoarece oamenii, caii şi nuferii nu sînt instanţieri ale clasei VIEŢUITOARE). Altfel spus, nu există instanţieri ale clasei VIEŢUITOARE, ci doar obiecte aparţinînd unor clase derivate din aceasta.

Revenind la limbajul C+ +, vom spune că, prin definiţie, funcţiile de tipul NUTRIŢIE se vor numi FUNCŢII PURE. Deci, o FUNCŢIE PURĂ este o METODĂ VIRTUALĂ ce nu poate fi DEFINITĂ (dar va fi obligatoriu DECLARATĂ) la un anumit nivel de abstractizare.

Odată definite FUNCŢIILE PURE, vom spune că o CLASĂ ABSTRACTĂ este o clasă care are cel puţin o metodă pură. O clasă de tipul VIEŢUITOARE va fi o CLASĂ ABSTRACTĂ tocmai fiindcă NU POT EXISTA INSTANŢIERI ALE EI. Acest lucru se datorează imposibilităţii de a defini funcţia de NUTRIŢIE.

Clasele abstracte se pot utiliza absolut identic cu cele ne-abstracte, cu unica diferenţă că NU POT EXISTA INSTANŢIERI ALE LOR.

Urmărind exemplul de mai jos, totul se va clarifica pînă în cele mai mici amănunte.

Exemplul 1

```
# include <process.h>
# include <dos.h>   // contine prototipul functiilor "sound",
                    //  "nosound" si "delay"

class joc{
 public :

// POST FIXAREA unei declaratii de metoda cu "=0" va determina
//  ca respectiva metoda sa devina PURA

 virtual void demo(void) = 0 ;     // FUNCTIE PURA
 virtual void melodie(void) = 0 ; // FUNCTIE PURA
};

class samurai : public joc{
 public :
    void demo(void)
    {
        system("samurai /d");
    }
```

```
    void melodie(void);
};

void samurai::melodie(void)
{
 for(int i=0;i<10;i++)
 {
    sound(1000+100*i); // provoaca emiterea unui sunet de
                       // frecventa egala cu valoarea para-
                       //·metrului actual
    delay(500);    // opreste executia programului timp de 500
                   // milisecunde (in acest mod se stabileste
                   // durata sunetului emis)
    nosound();         // opreste emiterea sunetului
 }
î

void main(void)
{
 samurai s;

 s.melodie();
}
```

De notat că dacă în declararea clasei "samurai" am fi avut secvenţa

```
class samurai : public joc
{
 public :
    void demo(void) = 0;
    //...
}
```

atunci "samurai" ar fi rămas o clasă abstractă, avînd însă o singură metodă pură ("demo"). Atenţie deci: în cadrul ierarhiei de clase, orice funcţie pură trebuie ori să fie redeclarată pură, ori să fie definită!!!

5.2. OVERLOADING OPERATORS

Termenul englezesc de "OVERLOADING OPERATORS" se referă la o facilitate specifică limbajului C++: dreptul programatorului de a adăuga operatorilor limbajului şi alte sensuri decît cele predefinite. Cu alte cuvinte, este vorba de o redefinire a operatorilor C++, fără ca vechile sensuri să se piardă.

Se ştie că operatorul "+" poate fi utilizat pentru a realiza adunări de numere întregi şi reale. Prin "OVERLOADING" (o traducere destul de plastică ar fi SUPRAÎNCĂRCARE), operatorul "+" ar mai putea fi folosit la adunarea de matrici, numere complexe sau raţionale. Cum s-ar putea realiza acest lucru? Foarte simplu: claselor MATRICE, COMPLEX sau RAŢIONAL li se va adăuga o metodă-operator numită "+". Evident, întreaga grijă pentru o utilizare corectă a operatorului cade în grija compilatorului. Astfel, presupunînd că am definit clasele COMPLEX şi RAŢIONAL că avînd un operator "+", în secvenţa de mai jos totul va decurge după cum dorim:

```
int        a=0,b=6;
double     c=4.5,d=6.7;
COMPLEX    e(2,3),f(3,4);
RATIONAL   g(4,5),h(5,6);

// .........

a=a+b; // adunare de intregi
c=c+d; // adunare de reali
e=e+f; // adunare de numere complexe
g=g+h; // adunare de numere rationale
```

Din punct de vedere al limbajului, operatorii redefiniţi nu sînt altceva decît nişte metode oarecare, purtînd însă nume de operatori (de exemplu metodele cu numele +, -, etc). Pentru a nu provoca erori de sintaxă, numele metodelor în cauză vor fi prefixate cu cuvîntul cheie "operator". Să considerăm următoarea secvenţă:

```
class matrice
{
// ...
 public :
    matrice& AdunaMatrici(matrice&);
    matrice& operator + (matrice&);
// ...
};
```

Unica diferenţă între cele două metode constă în comoditatea sporită a apelului celei de a doua:

```
matrice a,b,c;
// ...
c=a.AdunaMatrice(b); // INCOMOD
c=a.operator+(b); // CORECT, DAR LA FEL DE INCOMOD
c=a+b;            // MULT MAI COMOD SI MAI NATURAL
```

Înainte de a încheia această secţiune, vom mai face cîteva observaţii deosebit de importante. Deşi aproape oricărui operator i se poate da un nou sens, NICI UNUI OPERATOR NU I SE POATE SCHIMBA PRECEDENŢA SAU SINTAXA (un operator binar nu poate fi făcut unar şi nici invers)!!!!

De asemenea, nu este permis să se definească (introducă) noi operatori. E o restricţie absolut normală, lipsa ei putînd da naştere unor complicaţii atît de mari încît compilatorul să nu le mai poată rezolva.

Singurii operatori care nu pot fi redefiniţi sînt următorii:

. .* :: ?:

Reamintim că operatorul ".*" este utilizat la manipularea pointerilor la metode (vezi "3.4.").

ATENŢIE : directivele preprocesor # şi ## nu pot fi redefinite deoarece ELE NU SÎNT OPERATORI!

O altă chestiune importantă ţine de libertatea de redefinire a operatorilor. Pentru ca ea să fie totală, compilatorul NU VA FACE NICI O PRESUPUNERE LEGATĂ DE UN SENS PREDEFINIT AL OPERATORILOR. După cum sublinia şi Stroustrup ([STRO], p. 172), în încercarea de a oferi programatorului o maximă deschidere, compilatorul NU VA VERIFICA nici măcar în cazul redefinirii lui "=" DACĂ PRIMUL OPERAND ESTE SAU NU O "lvalue" (adică dacă i se pot atribui valori).

De asemenea, trebuie avut grijă la următorul aspect: în limbajul C expresiile

```
index++;
index+=1;
index=index+1;
```

erau echivalente. În C++, redefinirea lui "+" şi "=" nu e suficientă pentru ca acest lucru să fie adevărat. Trebuie ca programatorul să definească corespunzător TOŢI operatorii în cauză.

5.3. Operatori unari şi binari

Un lucru pe care nu l-am spus pînă acum este faptul că un operator poate fi nu doar metodă, ci şi funcţie "friend". Poate mai puţin aşteptată va fi IMPOSIBILITATEA de a defini un operator ca fiind METODĂ STATICĂ!!!

Dacă operatorul va fi definit ca metodă, există două posibilităţi:

este operator UNAR – în acest caz el nu va mai avea nici un argument

(operatorul unar are nevoie de un singur operand: obiectul care va apela metoda);

este operator BINAR – situaţie în care metoda va avea un singur parametru (care reprezintă cel de al doilea operand).

Dacă operatorul este o funcţie "friend", vom avea alte două posibilităţi, după cum acesta va fi:

UNAR – situaţie în care va avea un singur parametru;

BINAR – şi atunci va avea doi parametrii.

Tot legat de operatorii unari şi binari: fie "@" şi "#" cîte un simbol reprezentînd un operator unar, respectiv binar. Atunci, atît expresiile "a@" şi "@a", cît şi expresia "x#y" pot avea cîte două interpretări, respectiv

```
a.operator@();
operator@(a);
```

şi

```
x.operator#(y);
operator#(x,y);
```

după cum operatorii "@" şi "#" vor fi metode sau funcţii "friend". Dacă în vreunul din cazuri au fost declarate ambele forme, compilatorul va decide apelul corect în funcţie de ordinea parametrilor în cele două funcţii operator.

Trebuie să facem următoarea subliniere: orice operator care e funcţie "friend" a unei clase "B" trebuie să aibă CEL PUŢIN un argument de tipul "B" (instanţiere a clasei "B"). În caz contrar, ar exista riscul de a introduce ambiguităţi în cadrul expresiilor formate doar din termeni de tipuri predefinite. Unica excepţie de la această regulă o constituie operatorii "new" şi "delete".

Cu titlul de remarcă, amintim faptul că formele pre şi post fixate ale operatorilor redefiniţi "++" şi "--" nu vor putea fi deosebite de către compilator.

O altă observaţie, oarecum neaşteptată, este cea legată de faptul că deşi operatorii moştenesc SINTAXA (unar/binar) şi PRECEDENŢA operatorilor predefiniţi, ei nu vor moşteni şi COMUTATIVITATEA lor. Exemplul de mai jos va sublinia foarte bine acest lucru:

Exemplul 1

```
class exemplu_1{
    int a;
 public :
    exemplu_1(int a1) : a(a1) {}
```

```
    friend int operator+(int i,exemplu_1 e)
    {
        return i+e.a;
    }

    friend int operator+(exemplu_1 e,int i)
    {
        return i+e.a;
    }
};

class exemplu_2{
    int a;
 public :
    exemplu_2(int a1) : a(a1) {}

    int operator+(int i)
    {
        return i+a;
    }
};

void main(void)
{
 exemplu_1 e1(10);
 exemplu_2 e2(100);
 int a;

 a=e1+1;     // CORECT
 a=1+e1;     // CORECT

 a=e2+1;     // CORECT
 a=1+e2;     // ERONAT: ordinea parametrilor nu se potriveste
}
```

Ce se remarcă în acest scurt program? În primul rînd frapează faptul că, în ciuda aşteptărilor, compilatorul nu va ţine cont de comutativitatea adunării, şi va semnaliza în consecinţă o eroare în linia

```
a=1+e2;
```

În aceeaşi ordine de idei, dacă am elimina unul dintre cei doi operatori ai clasei "exemplu_1", am mai obţine încă o eroare de compilare.

Cea de a doua observaţie va evidenţia un avantaj al operatorilor "friend"

faţă de cei de tip metodă: nu se poate defini un operator comutativ care să fie metodă şi să opereze cu operanzi de tipuri diferite (deoarece implicit primul operand va fii întotdeauna obiectul care apelează metoda). În schimb, acest lucru va fi uşor realizabil prin mecanismul de funcţii "friend".

Vom prezenta acum încă două exemple de redefinire a unor operatori. În cîteva cuvinte, despre ce este vorba: vom defini o clasă "rational", care va simula o parte dintre operaţiile cu numere raţionale. În exemplele de mai jos s-au implementat operaţiile de înmulţire (operaţie binară cu simbolul "*") şi inversa (operaţie unară cu simbolul "~": se va inversa numitorul şi numărătorul, semnalizînd eventualele împărţiri cu 0). ATENŢIE: deşi operaţia de înmulţire este corect efectuată, nu se operează eventualele reduceri posibile între numitor şi numărător. În Exemplul 2 operatorii vor fi metode, iar în Exemplul 3 funcţii "friend".

Exemplul 2

```
# include <iostream.h>
# include <conio.h>

class rational{
    int numitor,numarator;
 public:
    rational(int,int);

    void tip(char *);

    rational& operator*(rational&);
    rational& operator~(void);
};

rational::rational(int n1,int n2)
{
 numarator=n1;

// NU SE ACCEPTA NUMITORUL 0
 numitor = n2!=0 ? n2 : 1;
}

void rational::tip(char * nume)
{
 int a;
 cout<<" Nr. rational   "<<nume<<" = "<<numarator
     <<"/"<<numitor<<"\n";
}
```

```
rational& rational::operator*(rational& n)
{
 rational * r;
 int n1,n2;

 n1=numarator*n.numarator;
 n2=numitor*n.numitor;

 r=new rational (n1,n2);

 return *r;
}

rational& rational::operator~(void)
{
 int n1,n2;
 rational* r;

 if(numarator!=0) {
// daca numaratorul este nenul se va efectua operatia
     n1=numarator;
     n2=numitor;
 }
 else {
// in caz contrar numarul este lasat neschimbat
     n2=numarator;
     n1=numitor;
 }

 r=new rational(n2,n1);

 return *r;
}

void main(void)
{
 rational a(1,2),b(3,4);

// stergere ecran
 clrscr();

 a.tip("a");
```



```
a=~a;
a.tip("a");

a=~a;
a.tip("a");

b.tip("b");

b=~~~~~b;
b.tip("b");

b=~b;
b.tip("b");

a=a*b;
a.tip("a");

b=a*b;
b.tip("b");

a=~a*~b; // ATENTIE: precedenta lui "~" este mai mare decit
         // cea a lui "*", deci "~" se vor executa inaintea lui "*"

  a.tip("a");
}
```

În urma execuţiei programului se vor afişa următoarele rezultate:

```
Nr. rational  a = 1/2
Nr. rational  a = 2/1
Nr. rational  a = 1/2
Nr. rational  b = 3/4
Nr. rational  b = 4/3
Nr. rational  b = 3/4
Nr. rational  a = 3/8
Nr. rational  b = 9/32
Nr. rational  a = 256/27
```

Ceea ce ar trebui totuşi evidenţiat este modul în care se efectuează operaţiile. Deoarece se doreşte o implementare cît mai reală, rezultatul nu va fi memorat în primul operand, ci va fi returnat ca entitate de sine stătătoare. Avînd în vedere această necesitate, rezultatul va fi păstrat într-o zonă de memorie alocată dinamic. Acest subterfugiu are două avantaje:

a. dacă rezultatul ar fi fost o variabilă locală metodei "operator *", valoarea

s-ar fi pierdut în urma părăsirii funcţiei în cauză;

b. dacă rezultatul ar fi fost păstrat într-o variabilă statică a metodei "operator*", am fi putut avea neplăceri în cazul unor înlănţuiri de operatori de genul "m=a*b*c*d";

După cum am mai spus, reluăm exemplul precedent, cu unica modificare că operatorii vor fi de astă dată funcţii "friend".

Exemplul 3

```
# include <iostream.h>
# include <conio.h>

class rational{
     int numitor,numarator;
 public:
     rational(int n1,int n2);

     void tip(char *);

     friend rational& operator*(rational&,
                                rational&);
     friend rational& operator~(rational&);
};

rational::rational(int n1,int n2)
{
 numarator=n1;
 numitor = n2!=0 ? n2 : 1;
}

void rational::tip(char * nume)
{
 int a;
 cout<<" Nr. rational   "<<nume<<" = "<<numarator
    <<"/"<<numitor<<"\n";
}

rational& operator*(rational& n,rational& m)
{
 rational * r;
 int n1,n2;

 n1=m.numarator*n.numarator;
 n2=m.numitor*n.numitor;
```



```
 r=new rational (n1,n2);

 return *r;
}

rational& operator~(rational& m)
{
 int n1,n2;
 rational* r;

 if(m.numarator!=0) {
    n1=m.numarator;
    n2=m.numitor;
 }
 else {
    n2=m.numarator;
    n1=m.numitor;
 }

 r=new rational(n2,n1);

 return *r;
}

void main(void)
{
 rational a(1,2),b(3,4);

 clrscr();

 a.tip("a");

 a=~a;
 a.tip("a");

 a=~a;
 a.tip("a");

 b.tip("b");

 b=~~~~~b;
 b.tip("b");
```

```
    b=~b;
    b.tip("b");

    a=a*b;
    a.tip("a");

    b=a*b;
    b.tip("b");

    a=~a*~b;        // ATENTIE: precedenta lui "~" este mai mare
                    // decit cea a lui "*"
    a.tip("a");
}
```

În secţiunile ce urmează vom prezenta, pe rînd, cîteva cazuri concrete de operatori a căror redefinire necesită o atenţie deosebită, datorată unui comportament mai puţin obişnuit. Cu excepţia operatorului "=" (vezi "5.6."), ABSOLUT TOŢI CEILALŢI se moştenesc conform regulilor obişnuite.

5.4. Operatorii "new" şi "delete"

Redefinirea acestor doi operatori implică mai multe restricţii:

a. "new" trebuie să returneze "void *" şi să aibă ca prim parametru un "size_t" (tip de dată definit în "stdlib.h" şi utilizat pentru manipularea lungimii zonelor de memorie destinate păstrării obiectelor);
b. "delete" trebuie să returneze "void", să aibă un prim parametru "void *" şi eventual un al doilea de tipul "size_t";
c. ambii operatori vor fi IMPLICIT şi OBLIGATORIU metode statice;
d. din precedenta condiţie reiese că "new" şi "delete" NU POT FI VIRTUALI.

Între atîtea restricţii, iată şi un fapt îmbucurător: operatorii standard "new" şi "delete" pot fi apelaţi atît în domeniul de vizibilitate al clasei ce-i redefineşte, cît şi în cel al altor clase derivate din aceasta. Acest lucru se poate realiza în două moduri: fie implicit (în caz că se crează/distrug obiecte de alt tip decît cel în cauză), fie explicit (prin formule de genul "::operator new" sau "::operator delete").

Vom da acum un exemplu foarte simplu, în care atît "new" cît şi "delete" vor fi redefiniţi.

Exemplu

```
# include <iostream.h>
```



```
class exemplu{
    int a,b,c;

// membrul "lungime" se va declara "static" pentru a putea fi accesat in
// cele doua metode statice "new" si "delete"
    static short lungime;

 public :
    exemplu(void)
    {
        cout<<"Apel constructor EXEMPLU\n";
        a=b=c=0;
    }

    ~exemplu(void)
    {
        cout<<"Apel destructor EXEMPLU\n\n";
    }

// se utilizeaza parametrul "lng" deoarece eventualele clase derivate din
// "exemplu" pot avea lungimi diferite de cea a clasei de baza
    void * operator new(size_t lng)
    {
       cout<<"Operatorul \"new\" redefinit !!!\n";
       lungime=lng;

// tipul argumentului (sir de caractere) este suficient pentru a specifica
// faptul ca se apeleaza operatorul standard "new"
        return (void *) new char[lng];
    }

    void operator delete(void * p)
    {
// se tipareste lungimea obiectului (in octeti)
        cout<<exemplu::lungime<<"\n\nOperatorul
           \"delete\" redefinit !!!\n";

// tipul argumentului (pointer la "char") este suficient pentru a specifica
// faptul ca se apeleaza operatorul standard "delete"
        delete[lungime] (char*) p;
    }
};
```

```
// variabilele membru statice trebuie initializate
short exemplu::lungime=0;

class derivat : public exemplu{
    double dbl;

 public :
    derivat(void)
    {
        cout<<"Apel constructor DERIVAT\n\n";
        dbl=12.34;
    }

    ~derivat(void)
    {
        cout<<"Apel destructor DERIVAT\n";
    }
};

void main (void)
{
 exemplu * ptr;
 derivat * ptd;

 ptr=new exemplu;
 delete ptr;

 ptd=new derivat;
 delete ptd;
}
```

Execuţia programului va determina tipărirea pe ecran a următoarelor date:

Operatorul "new" redefinit !!!
Apel constructor EXEMPLU
Apel destructor EXEMPLU

6

Operatorul "delete" redefinit !!!
Operatorul "new" redefinit !!!
Apel constructor EXEMPLU
Apel constructor DERIVAT

Apel destructor DERIVAT
Apel destructor EXEMPLU

14

Operatorul "delete" redefinit !!!

O observaţie importantă este legată de setarea valorii parametrului "lng": el nu va fi setat de către programator, ci va fi dedus DE CĂTRE COMPILATOR în momentul apelului lui "new". Valoarea sa va fi egală cu numărul de octeţi necesar pentru memorarea unui obiect de tipul dat. Prezenţa unui astfel de parametru este ABSOLUT NECESARĂ, deoarece metoda "new" va fi moştenită şi de clasa "derivat", ale cărei instanţieri au o altă lungime. Distrugerea obiectelor create dinamic se va face corect tot prin grija compilatorului.

5.5. Operatorul"()"

Redefinirea operatorului "apel de funcţie" ne aduce în situaţia de a crea un operator binar ne-static de forma

expresie(lista_de_expresii)

Utilizarea funcţiei-operator "()" are două mari avantaje:

a. EVALUAREA şi VERIFICAREA listei de argumente se face întocmai ca la o funcţie obişnuită.
b. Modul de funcţionare al mecanismului de apel. Deşi OPERATORUL este binar, FUNCŢIA-OPERATOR poate avea oriciţi parametrii (cel de al doilea operand fiind o listă de argumente). Deoarece lista de parametrii poate fi vidă, înseamnă că al doilea operand poate lipsi cu totul.

În exemplul de mai jos, am folosit operatorul "()" pentru implementarea unui iterator (funcţie de parcurgere a unei liste într-o ordine prestabilită).

Exemplu

```
# include <iostream.h>
# include <process.h>

struct elem
{
 int nr;
 struct elem * urm;

 elem(int n,struct elem * p) : nr(n) , urm(p) {}
```

```
};

// aceasta clasa este destinata pastrarii si manipularii de liste
// simplu inlantuite de elemente de tipul "elem"
class list{
 struct elem * cap,* curent;
// "cap" pastreaza un pointer la primul element al listei, iar
// "curent" pozitia la care s-a ajuns in cadrul operatiei de iterare

 public :
     list(void) { cap=NULL; }

     void init(int);

// prototipul celor doi iteratori
     struct elem * operator() (void);
     struct elem * operator() (int);
};

// creaza o lista cu "n" elemente
void list::init(int n)
{
 if(n<1)
 {
     cout<<"Initializare eronata !!!\n\n";
     exit(1);
 }

 for(int i=0;i<n;i++)
 {
     struct elem * p;
     p=new elem(i,cap);
     cap=p;
 }

// la inceput, pozitia curenta reprezinta primul element
 curent=cap;
}

// asigura o parcurgere ciclica a listei (in urma terminarii
// unei parcurgeri, se va incepe din nou cu primul element )
struct elem * list::operator() (void)
{
```



```
 struct elem * p=curent;

 curent = curent->urm!=NULL ? curent->urm : cap;
 return p;

}

// forteaza inceperea iterarii cu elementul avind numarul de
// ordine "pozitie"
struct elem * list::operator() (int pozitie)
{
 curent=cap;

// parcurge lista (eventual de mai multe ori, caci "pozitie" poate fi mai
// mare decit numarul de elemente din lista) si se pozitioneaza pe
// elementul dorit
 for(int i=0;i<pozitie-1;i++)
     (*this)();

 return (*this)();
}

void main(void)
{
 list l1;

 l1.init(4);

 for(int i=0;i<13;i++)
     cout<< l1()->nr << " ";

 cout<< "\n" << l1(12)->nr <<"\n";

 for(int j=0;j<13;j++)
     cout<< l1()->nr <<" ";

}
```

În urma execuţiei, pe ecran se vor afişa rezultatele:

```
3210321032103
0
3210321032103
```

5.6. Operatorul "="

Redefinirea operatorului ne-static "=" pune probleme cu totul deosebite. Acest fapt se datoreşte mai multor proprietăţi neîntîlnite la ceilalţi operatori.

Cu riscul de a ne repeta, vom începe prin a reaminti că tuturor claselor ce nu-l redefinesc pe "=" compilatorul le ataşează IMPLICIT o astfel de metodă. Ea va reprezenta un operator binar avînd ambii operatori de acelaşi tip. Acest operator atribuie tuturor variabilelor membru ale primului operand valorile membrilor corespunzători din cel de al doilea. Practic, va fi vorba de o copiere bit-cu-bit a tuturor variabilelor membru ale respectivei clase.

O a doua remarcă se referă la faptul că operatorul "=" NU ESTE NICIODATĂ MOŞTENIT DE CLASELE DERIVATE.

În fine, ori de cîte ori o clasă X are ca membru un pointer "p" pentru care se alocă dinamic memorie, trebuie avut grijă să nu se ajungă într-una din situaţiile de eroare de mai jos.

Exemplul 1

```
class ex
{
 int * pointer;

 public:
    ex(int i) { pointer=new int; (*pointer)=i; }
    ~ex() { delete pointer; }
};

void functie(void)
{
 ex e1(10),e2(20);

 ex e3=e2; // EROARE: constructorul va fi apelat doar pentru
           // "e2", iar destructorul si pentru "e2" si pentru "e3"

 e1=e2;  // EROARE: zona de memorie deja eliberata de doua
         // ori (de catre destructorii apelati pentru "e2" si "e3") va
         // mai fi eliberata inca o data de catre destructorul apelat
         // pentru "e1"
}
```

Acest exemplu evidenţiază subtila deosebire care există între operaţiile de "asignare" (se referă la un obiect CREAT DEJA) şi "iniţializare" (se referă la acţiunile ce au loc în cadrul constructorului).

Există două situaţii speciale, în care nu va fi utilizat operatorul "=", ci constructorul de copiere ("X::X(X&)"):

- transmiterea unor obiecte ca parametrii actuali ai unei funcţii;
- utilizarea obiectelor ca valori returnate de către o funcţie.

În cele ce urmează, vom relua exemplul din secţiunea precedentă, avînd grijă ca, de astă dată, să nu mai lăsăm la voia întîmplării asignarea şi iniţializarea.

Exemplul 2

```
# include <iostream.h>
# include <process.h>
# include <conio.h>

struct elem
{
 int nr;
 struct elem * urm;

 elem(int n,struct elem * p) : nr(n) , urm(p) {}
};

class list{
    struct elem * cap,* curent;

 public :
    list(void) { cap=NULL; }
    list(list&); // INITIALIZARE: constructor de copiere

    ~list(void);

    void init(int);

    void destruct(void);
    void construct(list&);

    struct elem * operator() (void);
    struct elem * operator() (int);

    void operator= (list& l) // ASIGNARE
    {
       if(this==&l) return; // ATENTIE la forme de
                            // genul "a=a"!
```

```
    // In situatii de genul "a = a", in urma distrugerii vechiului obiect,
    // reconstructia lui nu mai are sens !

        destruct();
        construct(l);
    }
};

list::list(list& l)
{
 construct(l);
}

void list::construct(list& l)
{
 struct elem * tmp=l.cap;

 cap=NULL;

// copiere de lista simplu inlantuita
 while(tmp!=NULL)
 {
    struct elem * p;

    p=new elem(tmp->nr,cap);
    cap=p;
    tmp=tmp->urm;
 }

// pozitionare pe inceputul listei
 curent=cap;
}

void list::destruct(void)
{
 struct elem * p=cap,* t;

// distrugere lista simplu inlantuita
 while(p!=NULL)
 {
    t=p;
    p=p->urm;
    delete t;
```



```
 }
}

list::~list(void)
{
 destruct();
}

// metoda "init" e identica cu cea din sectiunea precedenta
void list::init(int n)
{
 if(n<1)
 {
    cout<<"Initializare defectoasa !!!\n\n";
    exit(1);
 }

// crearc lista simplu inlantuita
 for(int i=0;i<n;i++)
 {
    struct elem * p;
    p=new elem(i,cap);
    cap=p;
 }

 curent=cap;
}

// ambii operatori sint identici cu cei din sectiunea precedenta
struct elem * list::operator() (void)
{
 struct elem * p=curent;

 curent = curent->urm!=NULL ? curent->urm : cap;
 return p;
}

struct elem * list::operator() (int pozitie)
{
 curent=cap;
 for(int i=0;i<pozitie-1;i++)
    (*this)();
```

```
 return (*this)();
}

void main(void)
{
 clrscr();

 list L1;
 L1.init(4);

 list L2,L3=L1;
 L2.init(6);

 L3=L2;
 L2=L1;

 cout<<"Lucram cu \"L2\" :\n";

 for(int i=0;i<5;i++)
    cout<< L2()->nr <<"\n";

 cout<<"Lucram cu \"L3\" :\n";

 cout<< L3(3)->nr <<"\n";

 for(int j=0;j<7;j++)
    cout<< L3()->nr <<"\n";
}
```

Rezultatele afişate de program vor fi cele de mai jos:

Lucram cu "L2" :
0
1
2
3
0
Lucram cu "L3" :
2
3
4
5
0
1

2
3

5.7. Operatorul "[]"

Termenul englezesc "subscript" desemnează operatorul "[]", care este un operator binar ne-static. Al doilea operand al său ("indexul") poate fi, practic, de orice tip. O linie de program de genul "a[b];" va fi interpretată ca "a.operator[](b);".

Vom da acum un exemplu interesant de utilizare a operatorului "[]". Ideea programului este foarte simplă. Se va crea un şir, ale cărui elemente conţin informaţii legate de cîte o persoană (în cazul nostru datele personale vor fi numele şi greutatea). Se doreşte identificarea persoanelor în cauză după trei criterii bine precizate (nume, greutate şi număr de ordine în cadrul şirului). În acest scop, au fost creaţi trei operatori de tipul "subscript", avînd indexul de tipul pointer la caracter, respectiv număr real sau întreg. Toţi aceşti operatori vor căuta în şir elementul descris de valoarea în cauză. Dacă este găsit, se va returna un pointer la el. În caz contrar se va returna valoarea NULL.

Exemplu

```
# include <iostream.h>
# include <string.h>

class exemplu;

class persoana{
     char   nume[20];
     double greutate;

     friend exemplu;

 public :

// initializeaza un obiect de tipul "persoana"
     void init(char * p,double g)
     {
         greutate=g;
         strncpy(nume,p,19);
         nume[19]='\0';
     }
```

```
// tipareste informatiile dintr-un obiect "persoana"
     void tip(void)
     {
     // se testeaza valoarea lui "this" deoarece "tip()" poate fi aplicat
     // asupra rezultatului unui operator "[]" (care poate fi NULL)
          if(this!=NULL)
            cout<<"\n\tNume : "<<nume<<
                  " Greutate : " <<greutate;
          else cout<<"\nElement inexistent !!!";
     }
};

class exemplu{
     persoana sir[3];

 public :
     exemplu(void);
     persoana * operator[] (char *);
     persoana * operator[] (double);

     persoana * operator[] (int i)
     {
    // testeaza apartenenta indexului la domeniul de valori corecte
          if (i<3) return & sir[i];
          else     return NULL;
     }
};

// initializeaza cele trei elemente ale lui "sir"
exemplu::exemplu(void)
{
 for(int i=0;i<3;i++)
 {
     char   n[20];
     double d;
     cout<<"\nTastati un nume si o greutate : ";
     cin>>n>>d;
     sir[i].init(n,d);
  }
}

// parcurge tot sirul cautind un element cu numele "string"
persoana * exemplu::operator[] (char * string)
```



```
{
 for(int i=0;i<3;i++)
     if(strcmp(sir[i].nume,string)==0)
        return &sir[i];

 return NULL;
}

// parcurge sirul cautind un element cu greutatea "greutate"
persoana * exemplu::operator[] (double greutate)
{
 for(int i=0;i<3;i++)
     if(sir[i].greutate==greutate)
        return &sir[i];

 return NULL;
}

void main(void)
{
 exemplu e;

 e["Marta"]->tip();
 e[70.5]->tip();
 e[0]->tip();

 e["barta"]->tip();
 e[-23.1]->tip();
 e[10]->tip();
}
```

În urma execuţiei programului, se vor afişa rezultatele de mai jos:

```
Tastati numele si greutatea : Fuicu 80.9

Tastati numele si greutatea : Marta 65

Tastati numele si greutatea : Popa 70.5
      Nume : Marta   Greutate : 65
      Nume : Popa   Greutate : 70.5
      Nume : Fuicu   Greutate : 80.9
Element inexistent !!!
Element inexistent !!!
Element inexistent !!!
```

Făcînd cîteva modificări, programul ar putea fi mai apropiat de o implementare reală. Deorece s-ar fi pierdut din claritea exemplului, respectivele modificări le vom propune doar ca temă. Astfel, ar fi util ca lungimea şirului de persoane să fie variabilă (setabilă printr-un parametru al constructorului). În plus, dacă operatorul "[]" nu găseşte numele sau greutatea în cauză, se va crea un nou element avînd caracteristica respectivă. Acest element va fi adăugat în lista de persoane. Recomandăm rezolvarea problemei de mai sus prin două metode:

a. utilizînd liste înlănţuite;

b. utilizînd şiruri create dinamic.

5.8. Operatorul "->"

O caracteristică deosebit de importantă a acestui operator ne-static constă în UNARITATEA lui. O expresie de genul "a->m" se va interpreta ca "(a.operator->())->m". Din acest motiv funcţia-operator va trebui să returneze fie UN POINTER la o instanţiere a unei clase, fie UN OBIECT de un tip pentru care este redefinit operatorul "->". Aceste observaţii sînt logice, si pot fi deduse din forma "(a.operator->())->m", după cum urmează: "(a.operator ->())" este un apel de funcţie, iar asupra valorii returnate de către această funcţie se va aplica "->m". Deci, acest ultim "->" este fie operatorul standard (caz în care funcţia-operator "->" trebuie să returneze un pointer), fie tot un operator redefinit (situaţie ce implică returnarea unui obiect).

În continuare, vom analiza un exemplu în care s-au implementat ambele tipuri de operatori.

Exemplu

```
include <iostream.h>

struct ex
{
 int membru;
};

class exemplul_1{
    struct ex * pointer;

public :
    void set(struct ex & p) { pointer=&p; }

    struct ex * operator -> (void)
    {
        return pointer;
    }
};

class exemplul_2{
 exemplul_1 * pointer;

 public :
    void set(exemplul_1 & p) { pointer=&p; }

    exemplul_1 operator -> (void)
    {
        return *pointer;
    }
};

void main (void)
{
 struct ex  s;
 exemplul_1 e1;
 exemplul_2 e2;

 e1.set(s);

 e1->membru=3; // ATENTIE: ESTE UN APEL AL FUNCTIEI
               // "exemplul_1::operator-> ()"

 cout<<e1->membru<<"\n";// se va tipari valoarea 3 (in urma
                        // unui alt apel al aceluiasi operator)

 e2.set(e1);
 e2->membru=5;//ATENTIE: SINT DOUA APELURI DE FUNC-
              // TII-OPERATOR ("exemplul_2::operator-> ()"
              // si "exemplul_1::operator-> ()" )

 cout<<e2->membru<<"\n";//se va tipari valoarea 5 (in urma inca
                        //unui apel al acelorasi doi operatori)
}
```

5.9. Conversii

Există unele situaţii în care "overloading"-ul nu poate rezolva toate problemele. Spre exemplu, dacă dorim o implementare serioasă a clasei numerelor raţionale, ne vom izbi de o serie întreagă de inconveniente. În primul rînd ar fi efectuarea întregului set de operaţii (+, -, *, /, etc) nu numai între numerele raţionale, ci şi cu valori de alt tip (de exemplu adunări, scăderi ş.a.m.d. între numere raţionale şi întregi). Datorită lipsei de comutativitate a operatorilor redefiniţi, fiecare funcţie-operator trebuie scrisă de două ori (pentru ca ambii operanzi să poată fi de ambele tipuri). Dar, de aici, rezultă o multitudine de funcţii-operator, care nu fac nimic altceva decît să diminueze sensibil eleganţa abordării problemei.

Neajunsul ar putea fi foarte uşor rezolvat dacă s-ar crea un operator care să realizeze o conversie implicită a operanzilor de tip întreg în "raţional". Evident că, de fapt, nu se pune problema particulară a conversiei din întregi în numere raţionale, ci una mult mai generală: conversia unui tip dat "T1" într-un altul "T2".

Această problemă se poate diviza în două subprobleme:

a. Conversia unui tip oarecare "T1" într-un altul "T2", DEFINIT DE CĂTRE UTILIZATOR. Această subproblemă se va rezolva printr-un constructor al clasei "T2", care va avea UN SINGUR ARGUMENT (de tipul "T1"). De notat că, în această situaţie, nu are nici o importanţă tipul lui "T1": el poate fi atît predefinit ("int", "double" etc), cît şi definit de către utilizator.
b. Conversia unui tip T1 definit de către utilizator într-un tip T2 predefinit. Această situaţie specială va fi tratată de către operatori de tipul "T1::operator T2(void);", care vor realiza conversia şi vor returna o valoare de tipul T2.

Dacă cei doi operatori de conversie sînt definiţi, ori de cîte ori în cadrul unei instrucţiuni se va impune una sau cealaltă dintre conversii, se va apela IMPLICIT funcţia în cauză. Dar atenţie: CONVERSIILE NU VOR FI EFECTUATE DECÎT DACĂ SÎNT UNICE!!! Şi încă o problemă: toate obiectele create prin conversii (indiferent ca sînt implicite sau explicite) vor fi distruse de îndată ce nu mai este nevoie de ele!

Să analizăm acum un scurt program care implementează ambele tipuri de conversii.

Exemplul 1

```
# include <iostream.h>

class rational{
    int numitor,numarator;

public :
    rational(int n1,int n2=1)
    {
// Acest constructor poate fi utilizat si la conversii implicite
// de genul "int" -> "rational".

        numarator=n1;
        n2!=0 ? numitor=n2 : numitor=1;
    }

// conversie"rational-real"
    operator double(void)
    {
        return numarator/(double)numitor;
    }

    void tip(char * nume)
    {
        cout<<nume<<" = "<<numarator<<"/"
            <<numitor<<"\n";
    }

    rational& operator * (rational& r)
    {
        return *(new rational(numarator*r.numarator
                              ,numitor*r.numitor));
    }
};

void main(void)
{
 rational a(1,2),b(2,3),c(3),d(7,9);
 double dbl;

 a.tip("a");
 b.tip("b");
 c.tip("c");
 d.tip("d");
 c=a*b;
 c.tip("c");
```

```
 d=b*c*3;    // conversie "intreg -> rational" : numarul intreg 3 se
             // va converti intr-unul rational prin apelul
             // constructorului "rational(3,1)";

 d.tip("d");

 dbl=d; // conversie "rational->real" : "d" se va converti intr-un real
 cout<<"dbl = "<<dbl;
}
```

Rezultatele programului vor fi afişate pe ecran, după cum urmează:

```
a = 1/2
b = 2/3
c = 3/1
d = 7/9
c = 2/6
d = 12/18
dbl = 0.66667
```

În acest moment ne putem permite o analiză mai serioasă a situaţiilor în care este de preferat utilizarea funcţiilor "friend" celor de tip membru. Vom începe prin a reaminti că, indiferent de context, constructorii, destructorii şi funcţiile virtuale TREBUIE să fie METODE.

Există însă o situaţie în care nu pot fi utilizate decît funcţiile "friend":

```
class exemplu{
 public:
    exemplu(char *);
    void METODA(void);
    friend void Functie_Friend(exemplu);
};

// ...

void main(void)
{
 char string[10];
// ...
 string.METODA(); // EROARE: nu se incearca o
                  // conversie implicita"char*" -> "exemplu"
 Functie_Friend(string); // CORECT: se incearca conversia
                         //  implicita "char*" -> "exemplu"
}
```


După cum reiese şi din comentarii, diferenţa între cele două apeluri constă în aceea că o METODĂ poate fi apelată doar de un OBIECT DEJA EXISTENT, pe cînd FUNCŢIILE "FRIEND" pot fi apelate şi de OBIECTE ÎNCĂ INEXISTENTE, dar CARE POT FI OBŢINUTE PRIN CONVERSII UTILIZATOR. În afara acestei situaţii deosebite (ţinînd de aplicarea conversiilor implicite) şi de cazul operatorilor redefiniţi care trebuie să fie comutativi, recomandarea de a folosi metode şi nu "friends" rămîne valabilă, chiar dacă uneori pare mai comodă utilizarea funcţiilor "friend".

Se impune o analiză a cazurilor în care folosirea conversiilor utilizator poate provoca apariţia unor ambiguităţi. Ele sînt sintetizate în exemplul de mai jos.

Exemplul 2

```
// STRUCTURI DE DATE NECESARE EXEMPLIFICARII
// POSIBILELOR ERORI
struct ex1{
 ex1(int i) { /* ... */ }
};

struct ex2{
 ex2(int i) { /* ... */ }
};

struct ex3{
 ex3(float d) { /* ... */ }
};

struct ex4{
 ex4(ex3 e) { /* ... */ }
};

struct ex5{
 ex5(ex4 e) { /* ... */ }
};

// FUNCTII NECESARE EXEMPLIFICARII POSIBILELOR
// ERORI
void f1(ex1 e)
{ /* ... */ }

void f1(ex2 e)
{ /* ... */ }
```

```
void f2(ex5 e)
{ /* ... */ }

void f3(ex4 e)
{ /* ... */ }

void f4(float d)
{ /* ... */ }

void f4(ex3 e)
{ /* ... */ }

void main(void)
{
 f1(1); // EROARE DE COMPILARE. EXPRIMARE AMBI-
        // GUA nu se stie daca e vorba de f1(ex1(1)) sau f1(ex2(1))

 f1(ex1(1));    // CORECT
 f1(ex2(1));    // CORECT

 f2(12.34); // EROARE DE COMPILARE: compilatorul nu e
            // asigure trei conversii succesive
            // ( "double"-> "ex3"-> "ex4"-> "ex5" )

 f3(12.34); // CORECT: compilatorul e capabil sa asigure doua
            // conversii succesive ( "double"-> "ex3"-> "ex4" )

 f4(12);    // CORECT: conversie standard "int"-> "float"
}
```

După cum se vede, ambiguităţile sînt semnalizate în faza de compilare. Ele apar în două situaţii:

a. În momentul în care conversia nu este unică. Există însă şi o excepţie: în caz că se poate efectua o conversie standard, toate conversiile definite de către utilizator vor fi ignorate. Din acest motiv nu se semnalizează eroare la apelul lui "f4(12)".
b. În cazul în care pentru a se ajunge la tipul de date cerut este necesară aplicarea IMPLICITĂ a mai mult de două conversii utilizator.

Şi o ultimă observaţie: deşi conversiile utilizator sînt posibile, nu trebuie uitat că folosirea lor duce la obţinerea unor programe ilizibile, şi, uneori, chiar neportabile.

5.10. Problemă propusă spre rezolvare

Ca exerciţiu, va sugerăm să realizaţi o implementare a unei clase "matrice", care să permită utilizarea matricilor de întregi de dimensiuni variabile (pentru analogie, vezi clasa "BASIC_ARRAY" din secţiunea "3.11."). Manipularea elementelor matricii se va face cu ajutorul operatorilor "[]". Un exemplu de utilizare a clasei "matrice" ar fi secvenţa de mai jos:

```
void functie(int NumarLinii,int NumarColoane)
{
 matrice m(NumarLinii,NumarColoane);//declarare matrice

 for(int i=0,j;i<m.get_NrLinii();i++)
 {
    for(j=0;j<m.get_NrColoane();j++)// bucla de afisare
                                    // a unei linii
    {
       m[i][j]=i*10+j;      // initializare element
       cout<<m[i][j]<<" "; // afisare element
    }

    cout<<"\n";    // trecere la rind nou
  }
}
```

În cazul în care nu reuşiţi să rezolvaţi problema, vă recomandăm să încercaţi o implementare a clasei "matrice" pornind de la următorul schelet de program:

```
class linie{
     int* pTablou; // pointer la un array de intregi

 public:

// metoda de mai jos returneaza o referinta la elementul din
// array avind numarul de ordine "index"
     int& operator[] (int index)
      { return pTablou[index]; }

// metoda "construieste_linie" creaza dinamic un array de
// "NrColoane" elemente
  void construieste_linie(int NrColoane)
 {
     pTablou=new int[NrColoane];
```

```
 }
};

class matrice{
     linie* pMatrice; // pointer la un array
     int    NrLinii,NrColoane;

 public :

// functia operator de mai jos returneaza o referinta la un array de
//  intregi care reprezinta linia "index" a matricii de intregi
     linie& operator[](int index)
     {
      return pMatrice[index];
     }

     int get_NrLinii(void)
     {
      return NrLinii;
     }

     int get_NrColoane(void)
     {
      return NrColoane;
     }

     matrice(int,int);
};

// constructorul de mai jos creaza dinamic o matrice de
// dimensiuni variabile
matrice::matrice(int nrLin,int nrCol)
{
 NrLinii=nrLin;
 NrColoane=nrCol;

 pMatrice=new linie[NrLinii];

 for(int i=0;i<NrLinii;i++)
    pMatrice[i].construieste_linie(nrCol);
}
```


6. STREAMURI

6.1. Introducere

În acest capitol ne vom abate oarecum de la spiritul cărţii, în sensul că vom aborda o temă ce ţine mai mult de bibliotecile C++, decît de limbajul propriu-zis. Procedăm astfel pentru că, de fapt, STREAM-urile reprezintă o bibliotecă de funcţii de intrare şi ieşire, organizată însă într-o ierarhie riguroasă de clase. În orice caz, o trecere a lor în revistă se impune cel puţin din două motive: pe de o parte este vorba de un set de facilităţi inexistente în C-ul clasic, iar pe de alta constituie un excelent exemplu de eficienţă şi eleganţă în crearea unei ierarhii de clase.

În cele ce urmează, nu vom prezenta prima versiune a bibliotecii de stream-uri C++, ci o versiune mai nouă: biblioteca IOSTREAM. Aceasta îi este superioară celei vechi prin capacitatea ei de a utiliza toate caracteristicile versiunii C++ 2.0 (între care şi moştenirea multiplă). Pe moment, spaţiul nu permite o analiză exhaustivă a tuturor problemelor legate de stream-uri. De aceea, vă recomandăm că, după aprofundarea chestiunilor prezentate în carte, să efectuaţi un studiu atent al tuturor header-elor produsului BORLAND ce conţin informaţii legate de stream-uri.

Care e scopul STREAM-urilor? Aşa cum am mai spus, este vorba de o modalitate de manipulare a funcţiilor de intrare şi ieşire. Cu alte cuvinte, biblioteca de stream-uri ne pune la dispoziţie un set de metode care permit înlocuirea clasicelor "scanf" şi "printf". De ce le-am prefera pe cele noi? Foarte simplu: pentru că sînt mai elegante, flexibile şi eficiente. Şi, pe deasupra, graţie mecanismului de redefinire a operatorilor ("overloading"), vom putea crea un set propriu de metode, care să MANIPULEZE şi să VERIFICE toate datele aferente unor operaţii de intrare şi ieşire la nivel de OBIECTE. Majoritatea metodelor pot fi folosite incluzînd în programe fişierul "iostream.h". Altele vor necesita includerea lui "strstream.h" sau "fstream.h" (ambele includ la rindul lor pe "iostream.h").

După atîtea preparative, să vedem în sfîrşit ce e un STREAM. Specialiştii firmei BORLAND îl definiesc astfel: un STREAM este un CONCEPT ABSTRACT, care înglobează ORICE FLUX DE DATE de la o SURSĂ la o DESTINAŢIE ([BOR2], pag. 166). Dacă vrem să privim problema mai puţin teoretic, vom înlocui termenul de "flux de date" cu mai concretul "operaţii de citire şi scriere", şi termenul de "concept abstract" cu expresia "totalitatea mijloacelor de realizare". Evident că, de la caz la caz, sursa şi destinaţia vor putea fi ecranul, tastatura, un fişier sau o zonă de memorie. Putem reformula deci definiţia după cum urmează: STREAM-urile reprezintă

totalitatea modalităţilor de realizare a unor operaţii de citire sau scriere.

Ierarhia de clase din biblioteca "iostream" are două clase de bază: "streambuf" şi "ios". Prima este destinată operaţiilor de intrare/ieşire utilizînd un format minimal de citire şi scriere. Cea de a doua clasă conţine variabile de stare, metode necesare interfaţării cu "streambuf" şi un set de facilităţi destinate tratării erorilor ce pot să apară în timpul operaţiilor cu streamuri. "Streambuf" va fi moştenită de clasele "strstreambuf" şi "filebuf", pe cînd "ios" va da naştere unei arborescenţe de moştenire ceva mai complicate:

```
class  istream : virtual public ios;
class  ostream : virtual public ios;
class iostream : public istream , public ostream;

class  istream_withassign : public istream;
class  ostream_withassign : public ostream;
class iostream_withassign : public iostream;
```

Prefixele "i" şi "o" se referă la INPUT şi OUTPUT, adică la tipul stream-ului (de intrare sau ieşire). Deci clasele "istream" şi "ostream" permit tratarea caracterelor extrase/depuse din, respectiv în, obiecte de tipul "streambuf". Mai ciudat este cazul lui "iostream", care, îmbinînd caracteristicile primelor două, permite operaţii bidirecţionale (în sensul că aceeaşi entitate poate fi folosită atît ca sursă, cît şi ca destinaţie). Diferenţa dintre fii (cei analizaţi pînă acum) şi nepoţii clasei "ios" (cei care au în nume "_withassign") constă în faptul că aceştia din urmă mai posedă un operator suplimentar de atribuire, pe care nu-l vom analiza aici.

Trebuie să subliniem că "cin" şi "cout" (vă reamintiţi instrucţiunile de genul "cout<<alfa;") nu sînt nimic altceva decît instanţieri ale claselor "istream_withassign", respectiv "ostream_withassign". NOTA BENE: aceste două obiecte reprezintă echivalentele lui "stdin" si "stdout".

Pentru înlocuirea lui "stderr", firma BORLAND defineşte două obiecte ("cerr" şi "clog"), ambele instanţieri ale clasei "ostream_withassign". Unica diferenţă dintre ele constă în modul în care se execută "golirea" lor (în engleză "flushing mode"): în cazul lui "cerr" după fiecare inserare, la "clog" doar la umplerea buffer-ului. Pentru a clarifica această problemă, vom da o explicaţie mai amanunţită. Presupunem că fiecare dintre cele două obiecte are cîte un buffer intern, care conţine caracterele ce se vor afişa. În cazul lui "cerr", operaţiunile decurg astfel: la fiecare tentativă de scriere textul va fi depus în buffer, şi, indiferent că-l umple sau nu, va fi afişat. În cazul lui "clog", afişarea nu se va efectua decît dacă bufferul este plin. Mesajele de o lungime mai mică decît cea a buffer-ului nu vor fi tipărite singure, ci doar împreună cu alte mesaje ce vor veni după ele, pînă la umplerea buffer-ului.

6.2. Operaţii de intrare şi ieşire

Avînd în vedere similarităţile existente între "cin" şi "cout", vom începe tratarea lor în paralel, subliniind diferenţele de cîte ori va fi nevoie.

6.2.1. Introducere

Stream-urile de intrare şi ieşire redefinesc operatorii de shiftare ">>" şi respectiv "<<". În noua accepţiune, ei vor purta numele de EXTRACŢIE, respectiv INSERŢIE. Ambii sînt operatori BINARI, avînd drept prim operand (în partea stîngă) un obiect de tipul "istream" (respectiv "ostream"), cel de al doilea putînd fi orice tip pentru care operatorul a fost definit. Cei doi operatori sînt PREDEFINIŢI pentru tipurile de bază ale limbajului ("int", "char", "float" etc.) şi pot fi REDEFINIŢI pentru orice tip de date creat de către utilizator. Dacă redefinim ">>" sau "<<" ca avînd cel de al doilea operand de tipul "T" (tip definit de utilizator), operatorul se va numi "T extractor", respectiv "T insertor".

Analizînd secvenţa de mai jos, vom vedea un exemplu foarte simplu de aplicaţie utilizînd stream-uri:

```
char caracter;
cin>>caracter;
cout<<caracter;
```

Ce face această secvenţă de instrucţiuni? Va citi un caracter "caracter" din stream-ul de intrare (implicit tastatura) şi-l va scrie apoi în cel de ieşire (implicit ecranul).

Deoarece ambii operatori posedă proprietatea de asociativitate stînga şi, în plus, returnează o referinţă la primul lor operand (deci un "istream&", respectiv "ostream&"), ei pot fi înlănţuiţi. Acest fapt va permite o utilizare mult mai comodă a lor. ATENŢIE: nu se acceptă decît înlănţuiri de operatori de acelaşi tip ("<<" cu "<<" şi ">>" cu ">>"), nu şi înlănţuiri mixte ("<<" cu ">>")

Exemplul 1

```
# include <iostream.h>

void main(void)
{
 int a,c;
 float b;

 cout<<"Tastati un intreg si apoi un real !\n";
```

```
cin>>a>>b;
c=a+1;

cout<<"a+b="<<a+b<<" a&c="<< (a&c) <<"\n";
}
```

În urma execuţiei programului, pe ecran se vor afişa următoarele rezultate:

```
12345.6789
a+b=357.678894 a&c=12
```

Ce se întîmplă în acest exemplu: sînt citite două numere, unul întreg şi unul real, după care se vor tipări rezultatele a două operaţii aritmetice. Ultima instrucţiune a lui "main()" merita privită cu atenţie. Deoarece operatorul redefinit "<<" îşi păstrează precedenţa, trebuie avut grijă că, în cazul în care încercăm tipărirea unor expresii ce folosesc operatori cu o precedenţă mai mică decît a lui "<<", să utilizăm parantezele (în cazul adunării nu era nevoie, dar era obligatoriu la "a&c").

Se mai cuvin făcute două observaţii:

a. "cin>>a>>b;" este echivalent cu

```
(cin.operator>>(a) ).operator>>(b);
```

b. utilizarea operatorului de shiftare "<<" în cadrul unei operaţii efectuate cu "cout" se poate face incluzînd operaţia în cauză între paranteze

```
cout<<"a<<2="<< ( a<<2);
```

6.2.2. STREAM-uri şi tipuri de date predefinite

Deoarece în cadrul acestei subsecţiuni se va analiza în detaliu funcţionarea operatorilor "<<" şi ">>", discutarea lor se va face separat.

6.2.2.1. STREAM-uri de intrare

a. EXTRACTORI pentru tipuri de date numerice

În această categorie intră numerele întregi şi reale ("short", "int" şi "long" cu sau fără semn, respectiv "float" şi "double"). Deoarece ele vor fi tratate asemănător, le vom explica împreună. Înainte de a analiza modul de funcţionare al mecanismului de extragere, trebuie să dăm o definiţie. Toate acele caractere pentru care funcţia de bibliotecă "isspace()" returnează o valoare nenulă se vor numi CARACTERE SPAŢIU (sau DE TIP SPAŢIU). Din această categorie fac parte - între altele - blancul, '\t' şi '\n'.

Revenind, a extrage o valoare numerică înseamnă:

- Ignorarea tuturor caracterelor de tip spaţiu.
- Un test dacă primul caracter ne-spaţiu este cifră, semn sau, eventual, punct zecimal (la reali). În caz că această condiţie este îndeplinită, se va continua cu următorul pas. Altfel, se va semnaliza o eroare şi nu vor mai putea fi efectuate operaţii de intrare pînă ce nu se tratează respectiva eroare.
- Se extrag toate caracterele consecutive, pînă la întîlnirea unuia invalid pentru respectivul tip de dată.
- În cele din urmă, se va efectua conversia string-ului în cauză la tipul de date respectiv.

b. EXTRACTORI pentru caractere

În cazul tipului de date "char", o secvenţă de genul

```
char c;
cin>>c;
```

atribuie variabilei "c" primul caracter NE-SPAŢIU aflat în stream-ul de intrare.

Dacă se doreşte extragerea unui şir de caractere, se vor citi TOATE CARACTERELE NE-SPAŢIU întîlnite consecutiv în stream-ul de intrare, pînă se va ajunge la un caracter SPAŢIU. String-ului astfel obţinut i se va adăuga (la sfîrşit) un caracter NULL. Trebuie tratat cu atenţie cazul în care string-ul de la intrare depăşeşte lungimea şirului ţintă. Cu un minim de cunoştiinţe relativ la formatele de intrare, astfel de probleme se evită foarte uşor (vezi secţiunea "6.4.").

Înainte de a încheia această subsecţiune, mai trebuie făcută o subliniere. Să presupunem că, la o operaţie de citire, primul caracter ne-spaţiu din stream-ul de intrare va fi INVALID pentru tipul de dată citit. În acest caz, valoarea variabilei - ţintă va rămîne neschimbată! În plus, se va seta indicatorul "fail" (eşec) a lui "cin", fapt care va duce la imposibilitatea de a mai citi din acest stream pînă la resetarea indicatorului. Mai multe amănunte despre indicatorul "fail" vom vedea în secţiunea "6.3." (în care vom analiza amănunţit toate problemele ţinînd de tratarea erorilor apărute în cadrul operaţiilor cu stream-uri). Pînă atunci, un scurt exemplu reprezentativ pentru astfel de situaţii:

Exemplu

```
# include <iostream.h>
```

```
void main(void)
{
 int var1=9,var2=8;

 cout<<var1<<" "<<var2<<"\n";//se vor tipari cele doua valori

 cin>>var1;      // vom tasta litera "m", fapt ce va duce
                 // la setarea lui "fail"

 cin>>var2;      // deoarece este setat flagul "fail", nu se va mai
                 // putea efectua citirea lui "var2"

 cout<<var1<<" "<<var2<<"\n"; // se vor tipari aceleasi doua
                              // valori : 9 si 8
}
```

Pe ecran vor fi afişate informaţiile de mai jos.

```
98
m
98
```

6.2.2.2. STREAM-uri de ieşire

Tipurile de date predefinite pe care le acceptă implicit un stream de ieşire sînt: "char", "short", "int", "long" (cu sau fără semn), "float", "double", "long double", "char *" şi "void*". Tipurile numerice vor fi afişate folosind convenţiile implicite de la "printf()" (cu condiţia de a nu fi modificat formatul de afişare). Astfel, în secvenţa de mai jos, cele două modalităţi de tipărire vor avea acelaşi efect:

```
int a;
long b;
char c;
double d;

// ...

cout<<a<<" "<<b<<" "<<c<<" "<<d<<"\n";
printf("%d %ld %c %g\n",a,b,c,d);
```

Pointerul la caracter este utilizat la tipărirea string-urilor, iar "void *" la cea a variabilelor de tip "pointer" (afişarea se va face în hexazecimal).

```
char a[]="program";
```



```
void * p=a;

cout<<"String = "<<a<<"      "<<"pointer = "<<p;
```

Efectul execuţiei acestei scurte secvenţe va fi afişarea pe ecran a următorului mesaj:

String = program pointer = 87F0:0000

6.2.3. Alte metode de manipulare a STREAM-urilor

Clasele "istream" şi "ostream" mai dispun de cîte două metode (simetrice ca mod de utilizare), care fac acelaşi lucru ca şi operatorii "<<" şi ">>". Ele se numesc "get" şi "read", respectiv "put" şi "write", şi vor fi analizate în cele ce urmează.

Diferenţa majoră între aceste funcţii şi cei doi operatori prezentaţi anterior constă în faptul că setarea diverselor formate de lucru (citire/scriere cu format special) nu influenţează decît pe cei din urmă.

Fie următoarea secvenţă:

```
char a[10];

// ...

cout<<a;
cout.write(a,10);
```

Ţinînd cont de cele afirmate mai sus, s-ar putea ca cele două funcţii să aibă efecte diferite, în funcţie de formatul cu care va realiza scrierea operatorul "<<".

a. get

Clasa "istream" posedă nu mai puţin de 6 metode purtînd acest nume. În cele ce urmează, ne vom limita la a vă prezenta două dintre ele.

Analizînd prototipul de mai jos, vom deduce că această metodă citeşte cîte un caracter din stream-ul de intrare şi-l înscrie în bufferul indicat de primul parametru. Numărul maxim de caractere care se citesc este dat de valoarea celui de al doilea parametru decrementat cu 1. Ultimul parametru va specifica aşa-zisul caracter de sfîrşit (dacă în stream-ul de intrare se va întîlni acest caracter, citirea se va opri chiar dacă încă nu s-a citit numărul maxim de caractere). ATENŢIE: caracterul de sfîrşit nu va fi pus în bufferul destinaţie şi nici nu va fi eliminat din stream-ul de intrare.

```
istream& istream::get(char *,int,int='\n');
```

Metoda "get" are şi o variantă mai simplă, prin care se citeşte un singur caracter. ATENŢIE: spre deosebire de citirea caracterelor cu ">>", în secvenţa

```
char c;
cin.get(c);
```

variabila "c" va primi valoarea PRIMULUI CARACTER din stream-ul de intrare, CHIAR DACĂ ACESTA ESTE DE TIPUL SPAŢIU.

b. **put**

Metoda simetrică lui "get" are următorul prototip:

```
ostream& put(char);
```

Scopul ei este de a realiza tipărirea unui singur caracter. Astfel, o secvenţă de genul

```
char c1='i';
int  c2='j';
// ...
cout.put(c1);
cout.put(c2);
```

va avea ca efect afişarea celor două caractere ('i' şi 'j').

c. **read**

Această funcţie oferă posibilitatea de citire a unui şir de caractere. Prototipul ei este următorul:

```
istream& read(char*, int);
```

Cu alte cuvinte, metoda "read" va avea doi parametrii: un pointer la un şir de caractere (care va desemna bufferul destinaţie) şi un întreg care precizează numărul de caractere de citit.

d. **write**

Metoda "write" permite scrierea unui şir de caractere (de lungime cunoscută) într-un stream, şi are următorul prototip:

```
ostream& write(char*, int);
```

Analizînd secvenţa de mai jos, totul pare banal.

```
char a[]="string";
cout.write(a,6);
```

Dacă furnizăm corect cei doi parametrii (pointerul la string şi numărul de caractere de tipărit), mecanismul merge strună. Totuşi, se impune o

observaţie: spre deosebire de operatorul "<<", "write" va tipări TOATE cele "N" caractere, chiar dacă în cadrul stringului va întîlni un caracter NULL.

Deşi mai există şi alte metode ale clasei "istream", în încheierea subsecţiunii, vom mai prezenta doar funcţia membru "putback()". Ea are un singur parametru, care este de tipul "char". După cum indică şi numele, metoda "putback" va încerca să pună în stream-ul de intrare caracterul primit drept parametru. În caz de nereuşită, eroarea va fi semnalizată prin setarea stării stream-ului la valoarea "fail" (vezi secţiunea "6.3.").

Metoda "putback" este deosebit de utilă în cazul în care trebuie citite de la tastatură toate caracterele consecutive care îndeplinesc o anumită condiţie. În astfel de situaţii, oprirea se va face în urma extragerii unui caracter necorespunzător, care nu va fi prelucrat în faza respectivă. Dar asta înseamnă fie că respectivul caracter trebuie repus în stream-ul de intrare, fie că el se va pierde. Dacă, în viitor, va fi necesară reanalizarea lui, pentru a nu-l pierde, se va folosi funcţia membru "putback".

Exemplul de mai jos subliniază exact cele spuse înainte. Funcţiile "CONDITIE1" şi "CONDITIE2" reprezintă criteriul de oprire din căutare (testeaza apariţia caracterului terminal). Procedura "extrage_string" va prelua cel mult "max" caractere consecutive ce îndeplinesc o condiţie dată.

Exemplu

```
# include <iostream.h>

typedef int (*pFct)(char);

// testeaza aparitia unui "#"
int CONDITIE1(char c)
{
 return c!='#';
}

//testeaza aparitia unui "&"
int CONDITIE2(char c)
{
 return c!='&';
}

// extrage din "cin" toate caracterele ce respecta o conditie
void extrage_string(pFct CONDITIE,
                    char * string,int max)
{
 char c;
```

```
int contor=0;

cin>>c;          // citirea primului caracter ne-spatiu

if(CONDITIE(c))
do
    {
        string[contor++]=c;
        cin.get(c); // citire caracter (poate fi si spatiu)
    }
while( CONDITIE(c) && contor<max );

string[contor]='\0';

cin.putback(c);
}

void main()
{
 char a[100];
 extrage_string(CONDITIE1,a,99);
 cout<<a<<"\n";

 extrage_string(CONDITIE2,a,99);
 cout<<a<<"\n";
}
```

6.2.4. Extractori şi insertori definiţi de utilizator

În cele ce urmează, vom da un exemplu de redefinire a operatorilor "<<" şi ">>" astfel încît să permită citirea şi scrierea de obiecte definite de către utilizator.

```
# include <iostream.h>

# define nl      "\n"
# define MAXIM    20

class exemplu{
     char   nume[MAXIM];
     int    virsta;
     double greutate;
```



```
 public :
     friend istream& operator>>(istream&,exemplu&);
     friend ostream& operator<<(ostream&,exemplu&);
};

istream& operator>> (istream& i,exemplu& e)
{
 i.read(e.nume,MAXIM-1); // se evita citirea a mai mult de
                         // "MAXIM-1" caractere

 e.nume[MAXIM-1]='\0'; // sfirsitstring

 i>>e.virsta;
 i>>e.greutate;

 return i;
}

ostream& operator<< (ostream& o,exemplu& e)
{
 o<<"\n\n FISA PERSOANALA  :  \n";
 o<<"\tNume      : "<<e.nume<<nl;
 o<<"\tVirsta    : "<<e.virsta<<nl;
 o<<"\tGreutate : "<<e.greutate<<nl;

 return o;
}

void main()
{
 exemplu el;

 cin>>el;
 cout<<el;
}
```

Evident că, într-un exemplu mai complet, functia ">>" mai poate avea încă două secţiuni:

a. Una de verificare a validităţii datelor de intrare (în sensul că vîrsta şi greutatea trebuie să fie pozitive, etc.).
b. Una de eliminare a "zgomotelor". De exemplu, dacă dorim ca formatul de intrare să fie de tipul

```
Nume : Istrate    Virsta : 24   Greutate : 70.5
```

trebuie eliminate din stream-ul de intrare string-urile "Nume :", "Virsta :" şi "Greutate :".

6.3. Tratarea erorilor apărute în operaţii cu stream-uri

Analizînd declaraţia clasei "ios", vom observa că există un set de indicatori de stare care păstrează toate informaţiile aferente erorilor apărute în cursul operaţiilor cu stream-uri. Aceşti indicatori sînt memoraţi la nivel de bit în cadrul variabilei membru "state".

```
class ios
{
// ...
 protected :
     int state; // pastreaza la nivel de bit valorile indicatorilor
                // de stare

 public :
      enum io_state
      {
         goodbit  = 0x00,// ultima operatie s-a efectuat corect

         eofbit   = 0x01, // semnalizeaza lipsa de caractere
                          // disponibile pentru citire

         failbit  = 0x02,// semnaleaza esecul ultimei operatii
                         // de intrare/iesire; stream-ul nu mai
                         // poate fi folosit pina ce flag-ul nu
                         // va fi sters

         badbit   = 0x04, // semnaleaza faptul ca ultima opera-
                          // tie de intrare/iesire a fost invalida;
                          // in urma resetarii flag-ului S-AR
                          // PUTEA ca stream-ul sa mai poata
                          // fi folosit

         hardfail = 0x08 // semnaleaza o eroare ireparabila
      };
};
```


Vom studia acum metodele de care dispune clasa "ios" pentru a trata erorile ce pot să apară.

Începem prin a aminti o primă familie de trei funcţii care nu au nici un parametru şi returnează un "int":

"eof": returnează o valoare nenulă dacă indicatorul "eofbit" este setat, şi 0 în caz contrar.

"fail": returneaz o valoare nenulă în caz că într-o operaţie de intrare/ieşire s-a înregistrat un eşec (este setat unul dintre flagurile "failbit", "badbit" sau "hardbit") şi 0 în rest.

"bad": funcţionează ca şi precedenţa, atît că semnalizează doar apariţia erorilor (deci nu ia în considerare flagul "failbit").

Metoda "good" nu are parametrii şi returnează tot un întreg: nenul dacă nici un flag de eroare nu e setat şi 0 în caz contrar.

Avînd acelaşi prototip ca şi "good", "rdstate" va returna starea curentă a flagurilor sub forma unui număr întreg. Altfel spus, această metodă va returna valoarea membrului "state".

În fine, "void clear(int=0);" setează variabila de stare "state" la valoarea parametrului actual. Pentru a seta un singur flag (lăsîndu-le pe celelalte nemodificate) se poate recurge la apelul :

```
my_stream.clear( my_stream.rdstate() | val_flag );
```

unde "val_flag" va lua una din valorile definite de tipul enumerat "io_state".

Este necesară o precizare deosebit de importantă: funcţia "clear" este capabilă să şteargă orice flag MAI PUŢIN "hardfail"!!!

Clasa "ios" redefineşte şi doi operatori unari: "!" şi "void *". Primul se comportă întocmai ca şi metoda "int fail(void)", iar cel de al doilea exact invers (returnează 0 în caz că unul din biţii în cauza e setat şi diferit de 0 în caz contrar). Graţie acestor doi operatori e posibilă scrierea unor secvenţe de genul:

```
char c;

while(cin>>c) // aici se utilizeaza IMPLICIT conversia
              // lui "istream" la "void *"
{
 cout<<c;

 if(!cout)     // nu este operatorul standard "!", ci cel redefinit
    cerr<<"Eroare la tiparire !!!\n";
}
```

În încheierea secţiunii, vom aminti că pentru scoaterea unui stream din starea de eroare este necesară îndeplinirea a două condiţii:

a. să dispară cauza ce a provocat eroarea;

b. să fie şterse flagurile ce semnalizează eroarea în cauză.

6.4. Operaţii de intrare şi ieşire cu format

6.4.1. Indicatorii de stare ai formatului

Pentru a înţelege mai bine problemele legate de formatele de citire şi scriere, vom arunca o privire asupra modului în care este declarată clasa "ios".

```
class ios{
// ...
 protected :
    long x_flag; // pastreaza (la nivel de bit) indicatorii de
                 // format
    int x_width; // numarul de caractere utilizate pentru
                 // afisarea unei valori pe ecran

 public :
    enum {
       skipws    = 0x0001, // ignora spatiile de la intrare

       left      = 0x0002, // la operatiile de iesire se face
                           // aliniere stinga

       right     = 0x0004, // la operatiile de iesire se face
                           // aliniere dreapta

       internal  = 0x0008, // aliniere dupa semn sau
                           // specificator al bazei

       dec       = 0x0010, // conversie in baza 10
       oct       = 0x0020, // conversie in baza 8
       hex       = 0x0040, // conversie in baza 16

       showbase  = 0x0080, // se va tipari si baza
       showpoint = 0x0100, // se va tipari punctul zecimal

       uppercase = 0x0200, // pentru afisarea literelor din
```



```
                           // cadrul numerelor reprezentate in baza 16 se vor
                           // utiliza majuscule

       showpos   = 0x0400, // intregii pozitivi prefixati de
                           // "+"

       scientific= 0x0800, // reali scrisi sub forma
                           // 9.99999E4
       fixed     = 0x1000, // reali scrisi sub forma 99999.9

       unitbuf   = 0x2000, // streamurile se "golesc" (afi-
                           // seaza) dupa fiecare inserare
       stdio     = 0x4000, // "stdout" si "stderr" se vor goli
                           // dupa fiecare inserare
    }; // SFIRSIT"ENUM"

// ...
}; // SFIRSIT "IOS"
```

La operaţiile de ieşire, este deosebit de important numărul de caractere utilizate pentru afişarea unei valori pe ecran. În scopul păstrării acestei caracteristici există variabila membru "x_width", pe care am descris-o în cadrul declaraţiei clasei "ios". Implicit, valoarea lui "x_width" este 0, ceea ce semnifică utilizarea unui număr minim de caractere, care să permită totuşi afişarea valorii în cauză. Există însă situaţii speciale (tabelări, rapoarte etc.), în care este necesară o tipărire pe coloane aliniate estetic. În acest scop se va utiliza metoda "width" cu prototipul

```
int ios::width(int);
```

Scopul ei constă în setarea lui "x_width" la valoarea parametrului actual. Metoda va returna continutul lui "x_width" inainte de apelul ei.

Dacă valoarea setată de funcţia "width" este insuficientă pentru afişarea informaţiei, se va utiliza valoarea implicită 0. În acest fel, se evită pierderile de informaţie rezultate din trunchiere.

Exemplu

```
int wdth=cout.width(25); // salvare "x_width" vechi si
                         // setare a noii valori la 25
cout<<"MESAJ 1";
cout.width(wdth);          // restaurare "x_width"
cout<<"MESAJ 2";
```

Trebuie amintit faptul că există şi o metodă cu prototipul "int width(void);". Ea returnează valoarea curentă a lui "x_width", fără a o mai modifica.

În legătură cu membrul "x_width", se mai cuvin făcute două observaţii:

a. În cazul în care valoarea lui este mai mare decît dimensiunea minimă necesară, se va efectua o aliniere la stînga sau la dreapta. Amănunte despre aceasta vom vedea în exemplul ce urmează.
b. După fiecare operaţie de scriere, "x_width" va fi setat implicit la 0. În cazul înlănţuirii de operatori "<<", valoarea iniţială a lui "x_width" nu va fi folosită decît PENTRU PRIMUL OBIECT TIPĂRIT. Celelalte se vor afişa cu valoarea implicită 0.

Variabila membru "x_flag" va păstra (la nivel de bit) indicatori ce stabilesc formatul de citire/scriere. Pentru setarea lor se va utiliza un set de metode definite în acest scop. Valorile din tipul enumerat vor fi utilizate ca parametrii ai metodelor respective. Trebuie să amintim că prin termenul englezesc de "format state flags" sînt desemnaţi indicatorii utilizaţi în stabilirea formatului de intrare/ieşire.

Vom prezenta acum cîteva metode care au drept scop stabilirea formatului de tipărire.

Exemplu

```
# include <iostream.h>

void main(void)
{
 double a=987.654;

 cout.fill('_');   // setare caracter de umplere
 cout.width(10);   // setare "x_width"

// setare aliniere stinga
 cout.setf(ios::left,ios::adjustfield);

 cout<<a<<"\n";        // afisare numar real

 cout.fill('>');    // setare caracter de umplere
 cout.width(15);    // setare "x_width"

//setare aliniere dreapta
 cout.setf(ios::right,ios::adjustfield);
```



```
 cout<<a<<"\n";        // afisare numar real
}
```

În urma execuţiei acestui scurt program, vom observa că, pe ecran, au fost tipărite string-urile

```
987.654___
>>>>>>>987.654
```

Putem sintetiza în cîteva cuvinte efectul celor două noi metode:

"fill": Are un argument de tip "char", care va deveni noul caracter de umplere. În caz că numărul de caractere necesare pentru afişarea unei valori este mai mic decît "x_width", spaţiile rămase goale vor fi umplute cu caracterul în cauză.

"setf": Are drept scop setarea unor flaguri. Primul parametru specifică noua valoare a indicatorului, iar cel de al doilea desemnează flagul ce urmează să fie modificat. Pentru a înţelege mai bine cum funcţionează "setf", este necesară descrierea a încă trei variabile membru ale clasei "ios":

```
class ios{
// ...

   public :
      static const long basefield;// "oct",
                                  // "dec", "hex"

      static const long floatfield;// "scientific",
                                   // "fixed"

      static const long adjustfield;
                              // "internal", "left" si "right"
// ...
};
```

Cel de al doilea parametru al metodei "setf" poate fi una din cele trei variabile membru prezentate mai sus. În dreptul fiecăreia din ele am trecut ca şi comentariu valorile pe care le poate lua.

6.4.2. Manipulatori

În exemplele precedente s-a văzut că modificarea caracteristicilor formatului ("x_width", flag-uri, caractere de umplere etc.) are un dezavantaj: nu permite înlănţuirea operatorilor "<<" care utilizează formate diferite. Eliminarea inconvenientului se poate face cu ajutorul unor funcţii speciale,

asemănătoare operatorilor, numite MANIPULATORI.

Vom relua acum exemplul precedent, în care înlocuim metodele obişnuite cu manipulatorii echivalenţi:

Exemplul 1

```
# include <iostream.h>
# include <iomanip.h>

void main(void)
{
 double a=987.654;

 cout<<setw(10)<<setfill('_')
     <<setiosflags(ios::left)<<a<<"\n"<<setw(15)
     <<setfill('>')<<setiosflags(ios::right)
     <<a<<"\n";
}
```

Efectul acestui program va fi absolut identic cu cel al precedentului, afişîndu-se aceleaşi string-uri:

```
987.654___
>>>>>>>>987.654
```

În cadrul secvenţei de mai sus, se poate deduce uşor scopul fiecărui manipulator (prin analogie cu primul exemplu). În continuare, vom încerca o scurtă sistematizare a categoriilor de manipulatori din "iomanip.h". Pentru aceasta îi vom împărţi în 5 grupe:

a. **dec, hex, oct**

Toţi cei trei manipulatori din această categorie se apelează fără nici un parametru.

Pot fi folosiţi atît la stream-uri de intrare, cît şi de ieşire. Prezenţa lor provoacă setarea flagului de conversie în baza 10, respectiv 16 sau 8.

b. **ws**

Nu are nici un parametru. Este specific stream-urilor de intrare, utilizarea lui declanşînd procesul de eliminare a caracterelor spaţiu.

c. **endl, ends, flush**

Nici unul din cei trei manipulatori nu se apelează cu vreun parametru.

Scopul lor este următorul:

- **"flush"**: goleşte (afişează) stream-ul;
- **"ends"**: inserează în stream un terminator de tip NULL;
- **"endl"**: afişează un stream, după ce în prealabil s-a inserat în el un caracter "newline".

d. **setiosflags, resetiosflags**

Au un singur parametru, de tip "long". Setează, respectiv resetează, flagul indicat de parametrul actual.

e. **setfill, setw, setprecision**

Au un singur parametru, de tip "int". Setează (la valoarea parametrului actual) caracterul de umplere, respectiv "x_width" sau numărul de cifre al părţii zecimale a numerelor reale. Se folosesc atît la stream-uri de intrare, cît şi la cele de ieşire.

Odată încheiată prezentarea manipulatorilor predefiniţi, vom relua Exemplul 1 pentru a arăta modul de creare a unui manipulator definit de utilizator:

Exemplul 2

```
# include <iostream.h>
# include <iomanip.h>

# define nl "\n"

ostream&
 manipulator_demonstrativ(ostream& o_stream)
{
 return o_stream<<nl<<nl<<"O noua entitate : "
                <<nl<<nl;
}

void main(void)
{
 double a=987.654;

 cout<<manipulator_demonstrativ<<setw(10)
     <<setfill('_')<<setiosflags(ios::left)<<a
     <<manipulator_demonstrativ<setw(15)
     <<setfill('>')<<setiosflags(ios::right)<<a;
}
```

În urma execuţiei programului, pe ecran se vor afişa mesajele:

O noua entitate :

987.654___

O noua entitate:

```
>>>>>>>>987.654
```

Utilizarea maipulatorilor este posibilă datorită faptului că există un operator redefinit "<<" avînd următorul prototip:

```
ostream& operator<< (ostream&
                (* PointerLaFunctie)(ostream&));
```

La fiecare folosire a unui manipulator, adresa acestuia va fi transmisă operatorului "<<" prin intermediul parametrului "PointerLaFunctie". În acest mod operatorul "<<" va putea apela funcţia în cauză.

6.5. Fişiere şi stream-uri

În această secţiune vom analiza modul de utilizare a stream-urilor în efectuarea operaţiilor de citire/scriere din/în fişiere. În acest scop, biblioteca de streamuri conţine trei clase speciale: "ifstream", "ofstream" şi "fstream". Ele sînt derivate din "istream", respectiv "ostream" şi "iostream".

Exemplu

```
# include <iostream.h>
# include <iomanip.h>

# include <fstream.h> // HEADERUL NECESAR
                      // LUCRULUI CU FISIERE

void eroare(char * pt)
{
 cerr<<"\n\nEROARE la incercarea de a deschide
    fisierul "<<pt;
 exit(1);
}

void copy(ofstream& destinatie,ifstream& sursa)
{
 char c;

 while( destinatie && sursa.get(c) ) // se folosesc cei
                                     // doi operatori
                                     // redefiniti(vezi "6.3.")
    destinatie.put(c);
}
```



```
void main(void)
{
 char sursa1[14],sursa2[14],destinatie[14];

 ifstream s1,s2;
 ofstream dst;

 cout<<"Introduceti numele celor 3 fisiere : "
    <<"Destinatie Sursa1 Sursa2\n\n";

 cin.read(destinatie,13);
 cin.read(sursa1,13);
 cin.read(sursa2,13);

 destinatie[13]=sursa1[13]=sursa2[13]='\0';

 s1.open(sursa1,ios::nocreate);
 if(!s1) eroare(sursa1);// utilizeaza operatorul redefinit "!"

 s2.open(sursa2,ios::nocreate);
 if(!s2) eroare(sursa2); // utilizeaza operatorul redefinit "!"

 dst.open(destinatie);
 if(!dst) eroare(destinatie);
 //s-a utilizat operatorul redefinit "!"

 copy(dst,s1);
 copy(dst,s2);

 s1.close();
 s2.close();
 dst.close();
}
```

Programul de mai sus încearcă să concateneze două fişiere ("sursa1" şi "sursa2"), păstrînd rezultatul într-un al treilea fişier ("destinaţie"). Din cîte se vede, modul de lucru este deosebit de simplu: se crează trei stream-uri ("s1", "s2", "dst"), care vor fi ataşate fişierelor corespunzătoare (prin metoda "open"). Folosind metodele deja cunoscute "get" şi "put", în funcţia "copy" se va realiza copierea unui fişier sursă în cel destinaţie. Cu ajutorul metodei "close", legătura logică dintre stream-uri şi cele trei fişiere va fi desfăcută.

În acest moment ar fi utile cîteva sublinieri:

a. Odată cu realizarea/desfacere legăturii stream-fişier, metodele "open" şi

"close" efectuează şi deschiderea (implicit în mod text) şi închiderea fişierelor în cauză.

b. În caz de eroare la deschiderea unui fişier, se va seta indicatorul de eroare al stream-ului respectiv (vezi "6.3.")

c. Fişierul odată închis, stream-ului i se va putea ataşa un alt fişier de lucru. Astfel, în exemplul nostru, secvenţa

```
// ...
dst.close();
dst.open(alta_destinatie);
// ...
```

ar fi întru totul corectă.

d. Dacă e cazul, metoda "open()" poate fi foarte uşor evitată graţie unui mecanism ce permite specificarea numelui fişierului ca parametru al constructorului stream-ului. De altfel, exprimarea

```
ofstream dst(destinatie);
```

este echivalenta cu

```
ofstream dst;
dst.open(destinatie);
```

În continuare, vom analiza mai pe larg metoda "open". Declaraţia ei în cele două clase "ifstream" şi "ofstream" este cvasi-identică:

```
void open(char* ,int=ios::out,
          int=filebuf::openprot); // OFSTREAM
void open(char* ,int=ios::in ,
          int=filebuf::openprot); // IFSTREAM
```

Primul parametru reprezintă un pointer la numele fişierului, iar al doilea este o valoare ce specifică modul de deschidere al fişierului. Lista tuturor valorilor posibile pentru acest al doilea parametru e definită în clasa "ios" sub forma

```
class ios{
// ...
 public :
    enum open_mode
    {
        in          = 0x01 , // fisier de intrare

        out         = 0x02 , // fisier de iesire

        ate         = 0x04 , // dupa deschidere, pozitia curen-
                             // ta va fi chiar sfirsitul fisierului
```



```
        app         = 0x08 , // deschis pentru apendare
                             // (scriere la sfirsit de fisier)

        trunc       = 0x10 , // daca fisierul exista, ii sterge
                             // continutul (optiune implicita in
                             // caz ca e vorba de un fisier de
                             // iesire si nu apare una dintre op-
                             // tiunile "ios::app" sau "ios::ate")

        nocreate    = 0x20 , // daca fisierul nu exista, se sem-
                             // nalizeaza eroare

        noreplace =   0x40 , // deschiderea esueaza daca
                             // fisierul exista

        binary      = 0x80     // deschidere in mod binar
    };
// ...

};
```

Sînt permise înlănţuirile specificatorilor modului de deschidere prin "SAU LOGIC" la nivel de bit. O linie de program de genul

```
ifstream f1("nume",ios::in | ios::nocreate);
```

este absolut corectă, şi va avea drept efect încercarea de a deschide pentru citire un fişier numit "nume". Încercarea va da greş în caz că fişierul nu există.

În fine, clasa "fstream" permite crearea unor stream-uri ce suportă atît operaţii de intrare, cît şi de ieşire. Altfel spus, graţie acestei clase, vor fi posibile secvenţe de genul:

```
fstream my_stream("filename.ext",
                  ios::in | ios::out);
streampos pozitie_citire,pozitie_scriere;
char c='b';
// ...
pozitie_scriere=my_stream.tellp();
my_stream.seekp(7,ios::beg);
my_stream<<c;
my_stream.seekp(14,ios::cur);
my_stream<<c;
// ...
```

```
pozitie_citire=my_stream.tellg();
my_stream.seekg(-9,ios::end);
my_stream>>c;
```

Ceea ce sare imediat în ochi, este prezenţa a două familii de funcţii-membru:

a. **"tellp", "tellg"**:

Returnează poziţia curentă de scriere ("Put") sau citire ("Get") în/din stream.

b. **"seekp", "seekg"**:

Servesc la modificarea poziţiei curente de scriere şi citire. Ele au un prim parametru întreg; acesta specifică numărul de caractere cu care se va modifica poziţia relativ la al doilea parametru (indică reperul faţă de care se va efectua modificarea). Valorile pentru al doilea parametru sînt definite în clasa "ios" dupa cum urmează:

```
class ios
{
//...
    public :
        enum seek_dir
        {
          beg = 0 , // inceput de fisier
          cur = 1 , // pozitia curenta in fisier
          end = 2   // sfirsit de fisier
        };
// ...
};
```

6.6. Probleme propuse spre rezolvare

1. Implementaţi un manipulator care să aibă drept scop testarea şi corectarea eventualelor erori ce pot apărea în cadrul operaţiilor de intrare /ieşire.
2. Redefiniţi operatorii "<<" şi ">>" de aşa manieră încît să implementeze operaţii de scriere şi citire prin intermediul interfeţei seriale a unui PC. Practic, cu ajutorul celor doi operatori, se vor putea efectua transmiteri de date (recepţie şi emisie) între două calculatoare.

ANEXA A

Lumea minunată a O.O.P.

Una dintre tendinţele cele mai interesante ale momentului constă în încercarea de a reuni "sub" o interfaţă grafică comună puterea mai multor O.O.L. ("Object Oriented Languages"). Astfel, din SMALLTALK se vor putea apela biblioteci scrise în C++ sau PASCAL obiectual. În acest sens trebuie subliniată iniţiativa IBM (orice s-ar spune, importanţa acestei firme este deosebit de mare) de a utiliza atît limbajul SMALLTALK cît şi C++. Avantajul ar consta în faptul că primul este folosit în aplicaţii informatice avînd interfeţe utilizator prietenoase (destinate în general unor utilizatori neinformaticieni), pe cînd al doilea "coboară" la nivelul aplicaţiilor legate direct de sistemul de operare (drivere de echipamente periferice, etc.).

Şi cum tot am ajuns în finalul prezentării, să aruncăm o privire sintetică asupra principalelor compilatoare obiectuale existente în acest moment (ianuarie 1992) pe PC-uri:

a. **TURBO C++**

DESTINAT:

- în general amatorilor, dar şi profesioniştilor dornici să dezvolte soft sub DOS

CARACTERISTICI:

- mediu de programare ieftin şi robust

PREŢ:

- aprox. $300

b. **BORLAND C++ 2.0**

DESTINAT:

- profesioniştilor

CARACTERISTICI:

- mediu de programare + TURBO DEBUGGER + TURBO ASSEMBLER + TURBO PROFILER + utilitar de conversie TURBO C 2.0 ->TURBO C++
- sistem evoluat de gestiune a memoriei în DOS
- poate exploata toate facilităţile WINDOWS; generează programe e-

xecutabile şi biblioteci de legare dinamică DDL specifice WINDOWS; are integrate toate funcţiile SDK Microsoft; posedă a o bibliotecă de sute de obiecte (inclusiv ferestre, meniuri, icon-uri, ...)

PREŢ:

- aprox. $800

c. **ZORTECH C++**

DESTINAT:

- de la amatori la profesionişti (există trei variante Standard, Developer şi Scientific)

CARACTERISTICI:

- asemănătoare cu BORLAND C++, dar NU ESTE MEDIU DE PROGRAMARE!!!
- ca organizare a ierarhiei de clase se apropie de SMALLTALK;
- versiunea Developer aduce primul compilator generînd cod specific unui I386-DX

PREŢ:

- aprox. $300 , $500 , $650

d. **SMALLTALK/V**

DESTINAT:

- în general profesioniştilor

CARACTERISTICI:

- mediu de programare
- limbaj pur obiectual, nu "obiectualizat" (ca C++)
- bibliotecă de 300 de clase conţinînd 97% din codul folosit pentru propria lui scriere (lipsesc doar rutinele de compilare şi gestiune a memoriei)

PREŢ:

- începînd de la $650 (în funcţie de versiune)

e. **TURBO PASCAL FOR WINDOWS**

DESTINAT:

- profesioniştilor (este fratele mai mare al lui TURBO PASCAL 6.0, destinat aplicaţiilor DOS)

CARACTERISTICI:

- mediu de programare "sub" WINDOWS
- bibliotecă de obiecte (TURBO VISION) mult mai completă decît BORLAND C++
- posedă ObjectWindows (utilitar de creare de interfeţe utilizator)

PREŢ:

- aprox. $380 (TURBO PASCAL 6.0: $250)

BIBLIOGRAFIE

1) [BOR1] BORLAND INTERNATIONAL
TURBO C++: Getting Started,
1990

2) [BOR2] BORLAND INTERNATIONAL
TURBO C++: Programmer's Guide,
1990

3) [ECK] Eckel Bruce
Using C++
Osborne/Mcgraw-Hill,
1990

4) [STRO] Stroustrup Bjarne
The C++ Programming Language
Addison-Wesley Series in Computer Science
1986

5) [MDLR] Canarache Ana, Breban Vasile
Mic dicţionar al limbii române
Editura Ştiinţifică
1974

6) Colecţia revistei "INFOPC", anul 1991